Experimental Support for Reconfigurable Application-Specific Accelerators

Isaac Gelado, Enric Morancho and Nacho Navarro
Computer Architecture Department. Universitat Politècnica de Catalunya
C/ Jordi Girona 1-3, Campus Nord. 08034 Barcelona, Spain
e-mail: {igelado,enricm,nacho}@ac.upc.edu

***Abstract*— New computer architectures are being proposed and will be implanted in the next few years. A common trend to improve the system performance is to include some reconfigurable logic components into future multi-core chips. Several prototypes already exist but there still is a lack of support from the programming models, the compilers and the operating system.**

Reconfigurable architectures enable a new execution model based on hardware accelerators. There have been several efforts in the last few years to extend the thread abstraction to include the characteristics of these new reconfigurable elements. However, since the thread abstraction was conceived to abstract the CPU, the inherent model is different from what hardware accelerators should expect.

In this paper we present a new way of managing hardware accelerators. We propose a new programming and execution model for hardware accelerators based on the *processing unit* abstraction. We study the relationship between hardware accelerators and the processor architecture, exploring the features that the new reconfigurable architectures should provide to the operating system. We also show how a prototype hardware accelerator achieves a 1.78x speed-up over the software implementation of the same functionality, out of a 41x potential speed-up, due to the data communication overhead. Finally we show our work on dynamic partial reconfiguration and how it enables using the *processing unit* abstraction to provide *virtual reconfigurable logic*.

I. Introduction

The Instruction Level Parallelism (ILP) current processors are able to exploit is reaching its limit. Therefore, computer architecture research has shifted to exploit new kinds of parallelism. For example, current Simultaneous Multi-Threading processors (SMT) try to exploit the existing Thread Level Parallelism (TLP) [1]. Following this trend, recent proposals suggest including additional computational elements besides the CPU. These components could be either additional processing units, as in the Cell processor designed by IBM, Sony and Toshiba [2], or reconfigurable logic, as in the Virtex-II Pro from Xilinx [3].

Currently, there is a lack of adequate support for these new computational paradigms from both the compilers and the operating system. There has been some work on extending the thread abstraction in order to support hardware accelerators [4]–[7], whether they are implemented as fixed hardware or as reconfigurable logic. However, the lack of convenient programming models and run-time support, still avoids hardware accelerators to be widely adopted.

As we will discuss, the inherent nature of reconfigurable logic accelerators does not make threads the most suitable abstraction. Since hardware accelerators are mainly implemented as processing elements, we propose using a different abstraction to expose them to user applications. We will introduce the *processing unit* to abstract hardware accelerators and we will show that it simplifies the management of hardware accelerators while reducing the area the reconfigurable logic consumes. We will also describe how mapping hardware accelerators into the application address space simplifies the programming model and allows the operating system to use the existing virtual memory mechanisms to implement *virtual reconfigurable logic*.

The rest of this paper is as follows: in section II we will analyze the existing work on the operating system for reconfigurable architectures. Later, in section III we will develop the benefits of using the *processing unit* abstraction to manage reconfigurable accelerators and we will describe how it can be used to implement *virtual reconfigurable logic*. In section IV, we will describe the target platform used for the experimental work. Section V will expose the experimental work which validates the ideas exposed on section III. It will also describe the interaction between hardware accelerators and other hardware components. In this section we will set the main guidelines to improve the design of hardware accelerators and we will study what facilities the computer architecture should provide to the operating system. We will also describe the experimental work on dynamic partial reconfiguration in current reconfigurable architectures. Finally, in section VI, we will draw some final conclusions.

II. Related Work

Upcoming computer architectures will include additional computational elements besides one or more CPUs. Two likely possibilities currently exist. The first one is including additional processing elements, like in the Cell processor, and the second one is surrounding the cores with reconfigurable

This work has been supported by the Ministry of Science and Technology of Spain and by the European Union (FEDER) under contract TIN2004-07739-C02-01, the HiPEAC European Network of Excellence and the MARCO/DARPA Gigascale Systems Research Center (GSRC).

logic, like in the Virtex-II Pro chip. Both approaches are not exclusive, and it is highly probable that future multi-core chips include reconfigurable logic as well as additional processing elements. Although our proposals can be applied to both alternatives, in the present paper we show them prototyped in the reconfigurable one.

A. Terminology

The literature on operating system support for reconfigurable architectures commonly uses the term *reconfigurable hardware* to make reference to reconfigurable logic, but this is not strictly correct. Reconfigurable hardware technologies, for example based on *hybrid Hall effect* ferromagnetic MOS devices [8] or dual-gate nanoscale transistors [9], are able to modify the behavior of hardware components on the flight, whereas reconfigurable logic ones, as *Field Programmable Gate Arrays* (FPGAs), are based on a fixed hardware which is able to implement different logic structures. Currently it is being researched how to implement reconfigurable logic devices using reconfigurable hardware [8], however, no reconfigurable hardware device is still commercially available. Therefore, for correctness, the term *reconfigurable hardware* is not used in this paper.

B. Hardware Threads and Hardware Tasks

To the best of our knowledge, most of the research on operating system support for reconfigurable logic has been based on the extension of the thread abstraction as representation of reconfigurable logic accelerators. Hardware accelerators are commonly used to replace a piece of code, and even a whole program, therefore, they are abstracted as *hardware threads*[1] [4], [6] or *hardware tasks* [5], [7]. Both terms reference the same concept: an execution flow which runs on hardware accelerators. Although some authors make no difference between them [6], usually the term *hardware thread* refers to an execution flow that runs on one or more hardware accelerators but *hardware task* refers to a whole execution flow implemented in hardware. Due to this, scheduling has different meanings when talking about *hardware threads* or *hardware tasks*. For *hardware threads*, scheduling means sharing hardware accelerators among the different threads that use them, whereas for *hardware tasks*, it means sharing the reconfigurable logic among different *hardware tasks*. Despite the scheduling issue, a common analysis of the two approaches can be performed.

For the operating system, a thread is characterized by its state and its context. In the case of *hardware threads* and *hardware tasks*, the state indicates whether the thread is running, suspended or waiting for another thread to produce a result. Existing proposals force hardware accelerators to implement their state using a *hardware thread interface*. This interface includes several status and command registers, a state machine, which acts as a thread controller, and an interface logic [10]. The context stores the necessary information to allow suspending a thread and rescheduling it back at a later time. This context is only necessary when implementing preemptable threads [11]. In hardware threads, the context depends on the concrete implementation of the hardware accelerator and currently there is not a complete solution to this problem [5].

Hardware threads impose an underlying programming model in which user-level programs have, at least, one main software thread, which creates as many hardware threads as accelerators the program uses [6]. A major issue on this programming model is the synchronization among the different threads, which requires using locks, semaphores and other synchronization mechanisms. Software and *hardware threads* use them, therefore synchronization mechanisms must be implemented into the reconfigurable logic [10]. Supporting additional features, as message passing, into hardware threads complicates even more the hardware and the software implementation [11].

Although *hardware threads* and *hardware tasks* have shown to be working abstractions to deal with reconfigurable logic, they present several problems. First, they require implementing additional logic into the hardware accelerators (the hardware thread interface, the synchronization hardware mechanisms or inter-thread communication hardware) which complicates the implementation and increases the reconfigurable logic area usage. Second, they impose a complex multi-threaded programming model, so applications may require algorithmic changes in order to use these hardware accelerators.

However, the weakest point on *hardware threads* and *hardware tasks* is a design issue. A thread abstracts a Von-Neumman style processor, which is a sequential piece of logic which inherently has a context and is able to implement different functionalities depending on the code that it executes. Although the context of a thread is completely architecture dependent, it always include, at least, a program counter and general purpose registers. Hardware accelerators may or may not have a context, but they do not expose a program counter to be read or modified by any external entity. Hardware accelerators are customized logic, not necessarily sequential, that implements a fixed functionality. Therefore, due to its different nature, general purpose vs specific computation, it seems that using the same abstraction for general purpose processors and application-specific hardware accelerators is not the appropriate approach.

III. Processing Units

Hardware accelerators implement a specific functionality, process some data and return the result. Typically, they implement registers or memory which are mapped into the physical address space. We propose to abstract them as *processing units*. This abstraction exposes hardware accelerators to user-level applications in an exokernel-like way [12]. Each *processing unit* represents a hardware accelerator. The operating system grants or denies the access of processes to the hardware accelerator and maps them into the process virtual address

[1] *Hardware thread* on reconfigurable operating systems makes reference to a completely different concept from the one expressed on the computer architecture literature by this same term.

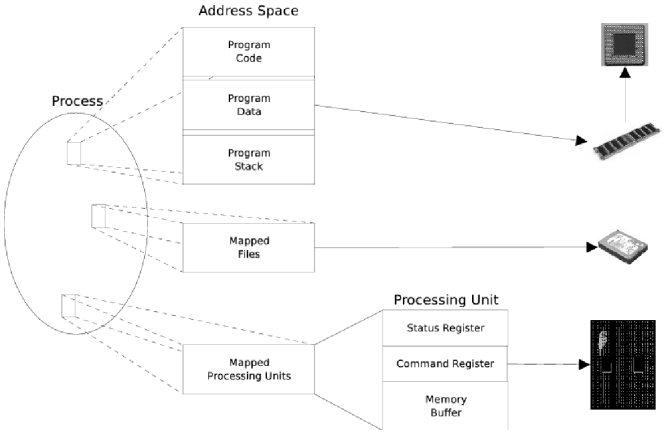

Fig. 1. An example of a memory mapped *processing unit*. In the process address space there is always present the memory containing the process code, data and the stack. Files and memory are usually mapped into the process address space, too. This mapping mechanism allows accessing to whatever hardware, for instance accelerators, using regular load/store instructions to certain memory addresses.

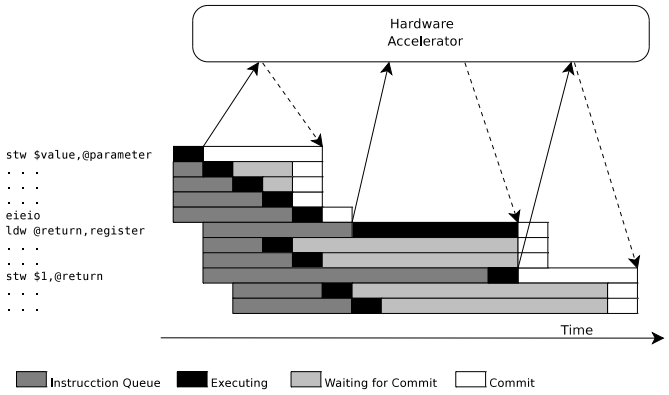

Fig. 2. An example of synchronization using load/store instructions. For simplicity, this example assumes an accelerator with a single input parameter and return value and an out-of-order processor which issues up to four instructions per cycle. In the left side the necessary code to use the accelerator. On the right side the time on each stage of the pipeline spent per each instruction. Fetch, decode and rename stages are not shown.

space. Then, applications have full access to the accelerator while the operating system guaranties that there will not be external interferences. Applications communicate with accelerators by means of the registers and memory implemented into the hardware accelerator. Both of them are accessible by virtual memory addresses, therefore regular load/store instructions are used to perform this communication.

In order to control which processes use the same hardware accelerator at the same time, the *processing unit* abstraction tracks the current owner. The abstraction includes access permission fields used to share an accelerator among different processes. Therefore, the owner of the accelerator can indicate to the operating system to which processes the access can be granted.

A prototype implementation of the *processing unit* abstraction has been implemented using the existing mapping mechanisms of an unmodified operating system. Figure 1 shows the hardware accelerator mapped into a user-level process, as any other memory region or file: mapping allocates a range of non-cacheable virtual memory addresses into the process address space and inserts the corresponding entries into the process page table. The process accesses to the accelerator registers and memory by reading and writing to the virtual addresses returned by the mapping operation and the *Memory Management Unit* (MMU) translates these virtual addresses to physical ones using the information provided by the page table.

Abstracting hardware accelerators as *processing units* provides a more flexible environment since it is not required to implement a defined status or a context, neither in hardware nor in software. Consequently, less logic is needed and, therefore, a better usage of the reconfigurable logic area is accomplished. But it remains possible to define a per-class standardized status register in such a way that similar hardware accelerators could be managed by the same code, like currently occurs for regular I/O devices.

A. Synchronization

Synchronization between the software and the hardware accelerators is necessary mainly:

- To start the computation into the accelerator when all the parameters have been sent.
- To read the return value when the accelerator has finished its computation.

As done on regular memory mapped devices, synchronization could be accomplished by means of polling or interrupts, depending on each specific implementation. Some bits in a register could share their meaning among hardware accelerators in such a way that the same synchronization code for all hardware accelerators is used and, therefore, generalization is not penalized.

Based on our experience of accelerators, we can propose a simplified synchronization mechanism by exploiting the usage of the load/store instructions that access the accelerator. Figure 2 shows how it works.

1) Store instructions (`stw`) communicate the input parameters to the accelerator. As soon as they are received, the accelerator starts computing the result.
2) A memory barrier instruction (`eieio`) ensures that the previous store will be executed before the next load.
3) A load instruction (`ldw`) gets the result from the accelerator. When the read request is received, the accelerator returns the result if it has been already computed. Otherwise, it delays sending the return value until it is ready.
4) A store instruction (to the same address used to get the return value) notifies the accelerator that it can overwrite the result with another computation.

Some interesting comments arise. First, store instructions are forwarded to memory during the commit stage [13]. Consequently, the input parameter is written into the accelerator

when there are no older instructions that could launch an exception or a branch misprediction that would flush the pipeline.

Second, the accelerator has to keep the result until the processor commits the load instruction. An interrupt or an exception could flush the pipeline and the load will be reissued, so if the accelerator releases the result previously, the user application would have not received the value. Therefore, it is necessary the last store instruction to release the result once the load has been committed.

There is at least one case where the computation could fail. If a hardware accelerator requires more than one input parameter, and the result depends on previous return values, and the computation starts each time an input parameter is written, then the result will not be correct. Several techniques, as check-pointing, can be used to solve this problem, however this is out of the scope of this paper.

As shown in figure 2, other independent instructions can be executed by the processor in parallel with the accelerator computation. Lots of opportunities arises for the compiler when using this synchronization mechanism, it could schedule independent instructions in between the load/store so the processor will exploit more ILP. We expect a leader role for the compiler on upcoming reconfigurable architectures. New generation compilers, as IMPACT, perform deep analysis and heavy transformations on data and code that will enable the optimization of the interface and the communication between the processor and the accelerators using mechanisms as the previously described one.

B. Virtual Reconfigurable Logic

There is off the shelf reconfigurable logic, such as the Xilinx Virtex-II Pro, which allows dynamic partial reconfiguration, which means to reconfigure a certain region of logic while the rest of the hardware remains working [3]. Thanks to other work we have done extending the Linux kernel, partial reconfiguration is available for the first time to the operating system. We propose implementing *virtual reconfigurable logic*: hardware accelerators are loaded on demand into the reconfigurable logic. In the meantime, the logic could contain a different hardware accelerator used by another process.

Since hardware accelerators are mapped into the process address space, the operating system uses the virtual memory mechanisms to map, protect and access the virtual reconfigurable logic:

- When the process maps a hardware accelerator into its address space, the operating system adds the necessary entries into the page table to make it accessible to the process. These entries will be set as not present as long as the hardware accelerator has not been already loaded into the reconfigurable logic.
- The first time the process tries to access the accelerator, a page fault exception is launched. The operating system catches this exception, looks for the mapped processing device location, asks the loader to load the accelerator, marks the associated entries in the page table as present and, finally, returns the control to the processor, which will reissue the faulting instruction.
- If needed, the operating system could decide to unload the accelerator from the reconfigurable logic in order to load a new one. In this case, it marks again the corresponding entries in the page table as not present.

Several problems must be solved for the loading/unloading of hardware accelerators while they are used by a process [5], however this issue is out of the scope of this paper.

C. Processing Units vs Hardware Threads

There are several differences between *processing units* and *hardware threads* that make the first ones a more suitable abstraction to work with hardware accelerators. The required time to create the data structure representing a *processing unit* is expected to be shorter than the one for a *hardware thread*. As shown previously, few data is necessary to implement a *processing unit*, however, a *hardware thread* uses the same data fields than a regular *software thread* plus the necessary ones to represent the hardware. Therefore, less data should be initialized when creating a *processing unit*.

The synchronization in *processing units* is less complex than in *hardware threads*. While the first ones only require adding a store instruction, *hardware threads* impose implementing additional hardware, which consume logic area. Because of the additional hardware, the programming model is more complex when using hardware threads since the programmer has to explicitly use inter-thread synchronization primitives, like getting locks or releasing semaphores.

The *processing unit* abstraction facilitates the loading on-demand of hardware accelerators because it maps them into the process virtual address space. Therefore, it is the MMU which detects when an accelerator is not present. However, *hardware threads* must relay on software policies to ensure the presence of the accelerator before using it.

IV. Experimental Environments

The *Xilinx University Program Virtex-II Pro Development System* [14] has been used as the target platform to develop all the experimental work. This board is based on a *Virtex-II Pro* chip which contains two *PowerPC 405* cores [15] surrounded by an FPGA.

Measurements have been taken using the internal clock cycle counter provided by the PowerPC core [15]. The necessary instructions to read this counter are added right before and after the code that is being measured.

A. Hardware Accelerator Performance

The logic for measuring the performance of hardware accelerators, when the *processing unit* abstraction is used, has been created using the *Xilinx Embedded Development Kit (EDK) 7.1* and the *Xilinx ISE 7.1* tools. Figure 3 shows the base hardware configuration, which uses a single PowerPC core running at 100MHz and implements into the FPGA the necessary devices (interrupt controller, DDR interface or bus arbiters) and three buses (a *Processor Local Bus* (PLB), a *On-Peripheral Bus*

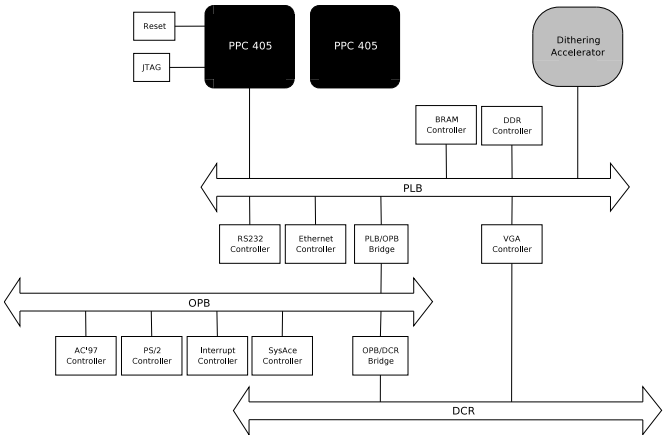

Fig. 3. System architecture of the Virtex-II Pro used for performance measurements. A single *PowerPC* core is used and the other one is isolated. Devices requiring low latency on data transmission are connected to the PLB whereas the other ones are attached to the OPB. The *VGA controller* is connected to the PLB and the DCR. The PLB connection allows accessing the device memory using regular load/store instructions and the DCR connection is used to work directly with the device registers. The *dithering hardware accelerator* is connected to the PLB and its registers are mapped into the system physical address space.

(OPB) and a *Device Control Register* (DCR) bus), all of them at 100MHz.

Our hardware accelerator prototype is connected to the PLB using the *Intellectual Property InterFace* (IPIF) core and implements two registers which are accessible from the main processor core. A port of *GNU/Debian* system running a *Linux PowerPC kernel 2.4.31-pre1* has been used as software platform.

For benchmarking, a *stereo audio linear dithering accelerator* has been developed in *Very High speed integrated circuit Description Language* (VHDL) to be used by audio applications. It is able to process the right and the left audio channels independently, consuming a single internal 25MHz clock cycle per channel (4 CPU clock cycles). This accelerator does not implement any status nor command registers, but processes each sample as soon as it is written into the corresponding input register. Since the *dithering* process subsamples a 32-bit signal into a 16-bit signal, the output from both channels are merged into a single output register.

B. Partial Reconfiguration

The partial reconfiguration of the *Virtex-II Pro* has been also measured as a previous step to implement *virtual reconfigurable logic*. Partial reconfiguration on Xilinx products is based on *Partial Reconfigurable Regions* (PRR) and *Partial Reconfigurable Modules* (PRM). A PRR is a portion of the reconfigurable logic which is suitable to be partially reconfigured. A PRM is defined as the logic module which can be partially reconfigured into a certain PRR. If a hardware accelerator has to be instantiated into two different PRRs, it must be implemented as two different PRMs. All the inputs and outputs of a PRR must go through special components, called *bus macros* [16].

The hardware configuration used for testing partial reconfiguration is very similar to the previous one. The DDR controller has been removed, the *dithering accelerator* has been replaced by a PRR and a *Internal Configuration Access Port* (ICAP) component has been attached to the OPB. This configuration has been generated using the *Xilinx EDK 6.3sp2* and a patched version, provided by Xilinx, of the *Xilinx ISE 6.1sp3*.

V. Experiments and Results

In this section we experimentally measure the improvement achieved by using the *processing unit* abstraction to use application-specific hardware accelerators. We also study current interconnection architectures used for communicating data between the main processor and hardware accelerators.

A. Experimental Work on Hardware Accelerators

A prototype application has been built to validate the *processing unit* abstraction. It is based on the existing mapping functionality provided by most operating systems. This prototype is also used to explore the relationship between hardware accelerators and other hardware components, as the processor or the memory. This experimental study will allow us to set the necessary guidelines to improve the design of hardware accelerators and establish what functionalities the computer architecture and the compilers should include to provide a better support to future reconfigurable architectures.

Madplayer, an MP3 player application, has been modified to use the *stereo audio linear dithering* hardware accelerator, described in section IV, instead of the default software implementation. At initialization time, the application maps the hardware accelerator into its address space using the `mmap()` system call. As a result, a global pointer to the accelerator registers is obtained. This pointer is used later to write successive samples to the *dithering* accelerator and to read the result. Before the application finishes, the `munmap()` system call is used to remove the accelerator from the application address space.

The *madplayer* application uses two nested loops to play a MP3 file:

- The outer loop reads one MP3 block from the file and decodes it. The decoding process writes the right channel samples into a buffer and the left channel samples into another one. This loop finishes when there are no more available MP3 blocks.
- The inner loop takes one sample from each buffer and applies the dithering processing to each one. This loop finishes when all the samples from the buffer have been processed.

The *dithering accelerator* is used by the inner loop and replaces two function calls to *original dithering* implementation. The number of CPU cycles consumed by the *dithering accelerator* and the software implementation have been measured as explained on section IV.

An MP3 file, with a length of 10.2 seconds, has been used as input, producing a total 452,742 measurements. As shown in Table I, the average number of CPU clock cycles for each

TABLE I

AVERAGE VALUE IN CPU CLOCK CYCLES REQUIRED TO PERFORM THE DITHERING PROCESSING ON ONE SAMPLE PER CHANNEL USING A HARDWARE ACCELERATOR AND THE ORIGINAL SOFTWARE IMPLEMENTATION.

Implementation	Memory and Computation	Only Computation
Hardware	457.68	191.04
Software	506.76	340.02

sample is 457.68 for the hardware implementation and 506.76 cycles for the software one. There is a difference of about 49 cycles between both implementations, which means a 1.11x speed-up. This little difference is because memory accesses to get the samples are included into the measurements. If the memory accesses are not included into the measured code, the average value becomes 191.04 for the hardware implementation and 340.02 for the software one. Now there is a difference of 149 cycles between both implementations, giving the hardware one a speed-up of 1.78x, which is still very far from the expected results.

The measurements presented in Table I show that accessing memory is more expensive when using hardware accelerators. The memory access costs 166.74 cycles for the software implementation while it costs 266.64 cycles when the hardware accelerator is used. The hardware implementation requires almost 100 more cycles to access to memory than the software implementation. The difference appears because in the software implementation the processor is able to overlap the memory-access latency with independent work. However, this scenario is not possible when using the hardware accelerator since the instructions that could be executed have been substituted by the accelerator. It is not possible for the operating system to hide this latency by scheduling another thread because its cost would be even higher.

B. Interconnection of Hardware Accelerators and the CPU

Hardware accelerators are connected to the main processors using a bus in the current reconfigurable platform used for the prototype implementation. To the best of our knowledge, so do other reconfigurable architectures. Therefore, it makes sense to develop a further analysis of the collected data in order to evaluate the adequacy of bus architectures to interconnect hardware accelerators and the processor in the future multicore chips.

Table II shows the average value and the typical deviation in number of cycles required by the accelerator into two columns. The first column considers the cost of accessing memory and the computation and the second one just considers the computation. If an interrupt arises while the processor is executing the measured code, the operating system takes control of the CPU in order to serve the incoming interrupt and, therefore, this measure will take a very high value. Consequently, the typical deviation is very high when all the data is considered. When memory accesses are included into the measured code, cache faults are also measured and the probability of arising an interrupt is higher. Therefore, in this case, the typical deviation takes a larger value.

TABLE II

AVERAGE VALUE, TYPICAL DEVIATION FOR DIFFERENT STATISTICAL ANALYSIS OF THE CLOCK CYCLES REQUIRED TO PERFORM THE DITHERING PROCESSING ON ONE SAMPLE PER CHANNEL USING A HARDWARE ACCELERATOR. THE FIRST ONE USES ALL THE COLLECTED DATA WHEREAS THE SECOND ONE OMITS THOSE MEASUREMENTS WITH A VALUE HIGHER THAN THE AVERAGE PLUS THE TYPICAL DEVIATION CALCULATED WHEN ALL THE DATA IS CONSIDERED.

	Memory and Computation		Only Computation	
	$E\{X\}$	σ	$E\{X\}$	σ
Full Data	457.68	5320.70	191.04	3546.41
Restricted Data	431.84	101.54	180.38	17.65

To avoid the interferences of the operating system, a new statistical analysis has been done without considering those values that are higher than a certain threshold. This value is the typical deviation when all data is considered. Up to 115 values are omitted from the measurements that consider the cost of accessing memory and 73 are skipped from the data collected when measuring only the computation time. The average number of cycles for the computation decreases as a consequence of not considering extreme values. So does the typical deviation, but its value is still very far from zero. It means that the number of CPU cycles required to write the data to the accelerator, to perform the computation and to read the result is not stable. Figure 4 shows a window of the histogram of the number of CPU cycles consumed per call to the hardware accelerator that confirms the previous numerical analysis. Most calls require 172 cycles, other ones take values from 191 to 238 cycles. This high variability is explained by the bus architecture used to interconnect the hardware accelerator and the processor, which is inherently non deterministic. Therefore, such architectures are not suitable to be used on real-time systems.

Current interconnection architectures mask the benefit of using hardware accelerators due to the high cost and variability of data communication. Our hardware accelerator takes 4 CPU cycles to compute each channel (a total of 8 cycles per input). Therefore, for this particular example, the 95.35% of the time is spent on data communication (considering 172 clock cycles as total time). Assuming that the hardware is able to perform the communication in one cycle, the accelerator produces a speed-up of 41x. A four cycle communication time means a 23x speed-up.

C. Lessons Learned on Hardware Accelerators

The experimental work has shown that the communication overhead is the important factor when using hardware accelerators. Current bus-based interconnection architectures introduce big latencies in data communication between the main processor and hardware accelerators and vice versa. Although

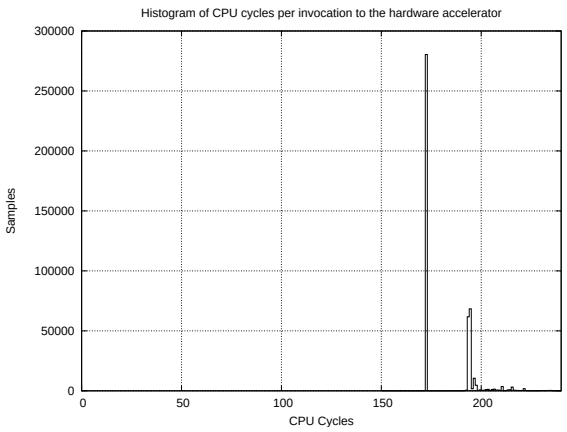

Fig. 4. Histogram of the number of CPU cycles per call to the hardware accelerator only considering those values smaller than 240. It shows that most of the calls take the same amount of cycles, however there are considerable variations.

a hardware accelerator is able to reduce the computation time by a huge factor, this improvement is hidden by the data communication overhead.

In order to minimize this overhead, more than one functionality or more complex functionalities could be implemented into the hardware accelerators, so the computation time would be bigger or comparable to the communication time. However, in one hand, implementing more functionalities is not always possible for all applications, and sometimes, the added functionalities will execute in hardware as fast as in software. In the other hand, complex functionalities usually require a bigger amount of data, so the communication time will increase too. In this case it is possible to use techniques, such as *burst transmission*, which reduce the number of cycles per byte during the data communication.

The cost of accessing memory is higher when using hardware accelerators because the ILP the processor is able to exploit decreases. Compilers could separate memory accesses from the accelerator calls in order to expose more ILP to the main processor. Moreover, those accelerators that require big amounts of data will employ, when available, *Direct Memory Access* (DMA) to get the data.

Finally, the architecture should provide additional support to make feasible the use of accelerators that perform fast fine-grained computations, as the *dithering accelerator* used as an example in this work. It should be possible to have a point to point connection between the accelerator and the main processor and, at the same time, make it accessible using memory addresses. This would reduce the communication overhead associated to the bus architecture while all the advantages of the *processing unit* abstraction would remain. However, bus architectures can still be used to interconnect computation intensive hardware accelerators, but the compiler must be aware of their characteristics to rearrange the code in such a way that more ILP is exposed to the processor so the processor will be able to execute independent instructions while the data is being transmitted.

TABLE III

TIMES IN MILLISECONDS CONSUMED BY EACH STAGE REQUIRED TO PARTIALLY RECONFIGURE A PROTOTYPE PRM.

Stage	Time	Total Time Percentage
Data Copy	24.23	93.40%
ICAP Commands	0.55	1.94%
Reconfiguration	1.21	4.66%

D. Partial Reconfiguration

A prototype stand-alone application has been used to measure the cost of partial reconfiguration. In this prototype, the hardware configuration loaded at boot time has the first PRM already instantiated into the PRR and a stand-alone application loaded into the system *Block Random Access Memory* (BRAM). The application loads a *partial bitstream* from a *compact flash* and reconfigures the PRR using the ICAP component. The reconfiguration process is done in three stages:

1) A block of the *partial bitstream* is moved from the main memory to a buffer provided by the ICAP component.
2) The adequate instructions are sent to the ICAP component using its memory mapped command registers.
3) The ICAP performs the actual reconfiguration of the FPGA.

Reconfiguration time has been measured as explained on section IV. Each PRM has a total length of 112,640 bytes and it is sent to the ICAP in blocks of 4,096 bytes.

Table III shows the time consumed in each previously described stage. Again, the communication time dominates over the other two. Sending the data to the ICAP component takes 93.40% of the time while the FPGA reconfiguration takes 4.66%.

First, the experimental results show that partial reconfiguration requires communicating big blocks of data and the reconfiguration time is very small compared to the data communication time. Therefore, partial reconfiguration process will also benefit of using DMA techniques.

Second, the operating system scheduler must be aware of the high number of cycles required to perform the partial reconfiguration. Scheduling policies should avoid selecting a thread to run if it is currently using an accelerator that is not present. The operating system should pre-fetch the loading of accelerators that will be used by the following processes on the run queue. There are more issues related to the impact of partial reconfiguration over the operating system policies, however they are out of the scope of this paper.

VI. CONCLUSIONS AND FUTURE WORK

In this paper the *processing unit* abstraction for reconfigurable architectures has been introduced. It allows a direct access to hardware accelerators from user-level applications and provides a simple programming model at the same time that it improves the reconfigurable logic area usage. It has been described a prototype implementation based on mapping

hardware accelerators into application address space. The guidelines for integrating this abstraction into the operating system have been analyzed too.

The *virtual reconfigurable logic* concept has been also introduced. It is based on the *processing unit* abstraction, the virtual memory subsystem and the partial reconfigurability feature of current reconfigurable logic. It allows the loading and unloading of hardware accelerators on demand in a simple way and using well-known operating system mechanisms.

A prototype implementation of the *processing unit* abstraction has been also described using the *madplayer* application and a *dithering accelerator*. This prototype has been used to measure the improvement of using hardware accelerators with regard to a software implementation. The obtained measurements have shown that the hardware implementation produces an speed-up of 1.78x over the software one. However, the results are not as good as expected due to the overhead introduced by the data communication between the main processor and the accelerator. It has been also shown that memory accesses are more penalized when hardware accelerators are used because there is less amount of ILP the processor is able to exploit.

In order to take profit of all the benefits of reconfigurable architectures, hardware accelerators have to perform a big amount of computation when they are connected to the processor using a bus. The compiler must be also involved by reordering the code in such a way that the processor could execute instructions while the input data is being read and the accelerator is performing the computations. When big amounts of data are required as input for hardware accelerators, they should obtain the data using DMA in order to allow the processor to perform other work while the data is being transmitted. Again, the compiler will have a leading role to accomplish this task by ensuring that the processor will have instructions to execute while hardware accelerators are computing results. Finally, new interconnection architectures should be provided by the processor in order to allow using highly optimized hardware accelerators as the *dithering* one, where the computation has been extremely reduced.

There are still several open research issues on operating system support for reconfigurable architectures. We are working on synchronization for hardware accelerators with multiple input values which will result on a particular implementation. Also the Linux kernel is being modified to incorporate the *processing unit* abstraction. This will be used as a prototype platform to implement *virtual reconfigurable logic* and to solve the problems related to the loading and unloading of stateful hardware accelerators.

Reconfigurable architectures introduce new challenges to the compilers and the operating system, but it needs that the computer architecture implements new mechanisms in order to build a more flexible execution environment as well as to improve the system performance.

ACKNOWLEDGEMENTS

We would like to thank the Xilinx University Program for its valuable help on partial reconfiguration and its hardware and software donations. We would also like to thank the IMPACT Research Group at the University of Illinois at Urbana-Champaign where this work was partially developed.

REFERENCES

[1] D. M. Tullsen, S. Engers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *Proc. of the 22th International Symposium on Computer Architecture*. Santa Margherita, Italy: ACM Press, Jan. 1995, pp. 392–403.

[2] D. Phams, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Jhons, J. Khale, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Wamock, S. Weitzel, T. Yamazaki, and K. Yazawa, "The Design and Implementation of a First-Generation CELL Processor," in *Proc. of the IEEE International Solid-State Circuits Conference*. San Francisco, USA: IEEE Press, Feb. 2005, pp. 184–185,592.

[3] *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, 4th ed., Xilinx Inc., San Jose, USA, Mar. 2005.

[4] G. B. Wigley and D. A. Kearney, "The First Real Operating System for Reconfigurable Computing," in *Proc. of the 6th Australian Computer Science Week (ACSAC)*. Gold Coast, Australia: IEEE Press, Jan. 2001.

[5] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Designing an Operating System for a Heterogeneous Reconfigurable SoC," in *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS'03)*. Nice, France: IEEE Computer Society, Apr. 2003.

[6] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden, "Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link," *IEEE Micro*, vol. 24, no. 4, pp. 42–53, July 2004.

[7] H. Walder and M. Platzer, "A Runtime Environment for Reconfigurable Hardware Operating Systems," in *Proc. of Field Programmable Logic*. Leuven, Belgium: Springer, Aug. 2004, pp. 831–835.

[8] S. P. Ferrera and N. P. Carter, "A Magnetoelectric Macrocell Employing Reconfigurable Threshold Logic," in *Proc. of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA'04)*. Monterey, USA: ACM Press, Feb. 2004.

[9] P. Beckett, "Towards a Reconfigurable Nanocomputer Platform," in *Proc. of the 7th Asia-Pacific Computer Systems Architectures Conference (ACSAC)*. Melbourne, Australia: ACS, Jan. 2002.

[10] R. Jidin, D. Andrews, and D. Niehaus, "Implementing Multi Threaded System Support for Hybrid FPGA/CPU Computational Components," in *Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithms*. Las Vegas, USA: CSREA Press, June 2004.

[11] C. Steiger, H. Walder, and M. Platzer, "Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1393–1407, Nov. 2004.

[12] D. R. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: an Operating System Architecture for Application-Level Resource Management," in *Proc. of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain, USA: ACM Press, Dec. 1995, pp. 251–266.

[13] G. S. Sohi and S. Vajapeyam, "Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors," in *Proc. of the 4th annual International Symposium on Computer Architecture*. Pittsburgh, USA: ACM Press, 1987, pp. 27–34.

[14] *Xilinx University Program Virtex-II Pro Development System. Hardware Reference Manual*, 1st ed., Xilinx Inc., San Jose, USA, Mar. 2005.

[15] *PPC405Fx Embedded Core. User's Manual*, International Business Machines Corporation, Hopewell Juction, USA, Jan. 2005.

[16] A. Dolin, "New tools for FPGA Dynamic Reconfiguration," The Future of Configurable Hardware Symposium, Xilinx Research, Ghent, Belgium, Dec. 2005.