Enabling Dynamically Reconfigurable Technologies in Mid Range Computers Through PCI Express

Charalampos Vatsolakis[12], Kyprianos Papadimitriou[12], and Dionisios Pnevmatikatos[12]

[1] School of ECE, Technical University of Crete
[2] Institute of Computer Science, Foundation for Research & Technology - Hellas
cvatsolakis@isc.tuc.gr, {kpapadim, pnevmati}@ics.forth.gr

Abstract. Efficient I/O access is crucial in reconfigurable hardware platforms for implementing high-performance systems. Such platforms can outperform CPUs and GPGPUs in executing applications characterized by inherent parallelism. However, the system-level performance depends heavily on sustaining high transfer rates for feeding data into the reconfigurable hardware and getting the results back to the end-user. In the present work we propose and implement a hybrid system comprising a host computer and an FPGA platform. The latter acts as co-processor into which hardware accelerators are loaded and executed in a transparent way, i.e. user is not involved in FPGA programming neither controlling its execution. Depending on the user request, the FPGA can be reconfigured either partially or entirely. Initially, we discuss the current state-of-the-art on I/O interfaces attached to FPGAs focusing primarily on the PCI Express (PCIe). Then, we present our system on which we implemented a design for measuring end-to-end throughput. We have developed a simple yet functional interface for serving the communication between software and hardware over PCIe v1.0 bus. At system-level, we achieved a throughput of 544 MBytes/s and 618 MBytes/s for DMA writes and reads respectively, over a PCIe four-lane (x4) connection. This includes all overhead such as communication delays and systems calls for requesting services from the operating system. Our work can be used as the basis for programming and executing hardware accelerators under the control of a run-time system.

1 Introduction

Desktop users usually possess a mid-range workstation with 2-4 processing cores and a limited amount of main memory. In general their needs regarding the type of applications they execute during workstation's life-time, concern applications such as word processors, spreadsheets, simple math calculations, drawing and so on. However, in some cases a user should execute compute-intensive applications such as image or signal processing; even more, the user might need to get the results back at real-time. Doing this using the host CPU is a poor option as

software execution may take from hours to days. Moreover, the CPU might not be able to withstand the processing load, thus resulting to system crash. Multithreading partially solves the problem but is not always sufficient. A viable solution is the use of GPGPUs that has been proven to be effective for highly parallelizable applications.

FPGA technology stands as a strong competitor against the aforementioned solutions and comes with an unprecedented advantage; it offers fine-grain hardware customization allowing to achieve the exact level of parallelism and pipelining needed per application to get the best possible performance. Hardware implementations of entire applications or compute-intensive algorithmic parts should be stored in a repository of IP cores. A run-time system residing on the software side loads the requested core to the FPGA for acceleration. Present works aims at addressing part of this functionality by providing an efficient system that communicates with and reconfigures an FPGA accordingly. During our work we had to resolve several issues such as the mechanism for FPGA reconfiguration and the interconnection between the CPU and the FPGA.

The Input/Output (I/O) of the FPGA can add considerable overhead to the total system execution. Although the computational power of FPGAs serves well kernel acceleration, the total execution time can be affected from the rate of servicing incoming/outgoing data. In cases where the FPGA is shared amongst multiple users, there are certain reconfiguration costs we need to take into account. Recent FPGA technology offers mechanisms allowing to reach high reconfiguration rates.

Efficient I/O has traditionally been an important research area and multiple interfaces have been developed over the years either parallel or serial. PCIe is a highly efficient serial interface. It is the successor of the parallel PCI interface, and there is a number of lanes available per device. The device throughput is related to the number of lanes a device supports. In terms of bandwidth, the initial version of PCIe (v1.0) offered a maximum of 2.5 Giga Transfers per second (GT/s) per lane for each direction. Due to the 8b/10b encoding, the maximum achievable bandwidth per lane was 250 MB/s [1]. The second version of the PCIe interface (v2.0), doubled the transfer rate up to 5.0 GT/s, leading to a throughput of 500 MB/s per lane for each direction [2]. The latest PCIe specification (v3.0) doubles the payload to 1 GB/s per lane by increasing the transfer rate up to 8.0 GT/s and using a more sophisticated 128b/130b encoding scheme [3]. Practically, each PCIe version doubles the per lane throughput of its previous, without necessarily doubling the clock frequency.

The scope of present work is to deliver an efficient reconfiguration mechanism and a high throughput I/O interface to desktop users. We envision a low-cost heterogeneous environment able to support the execution of multiple accelerators loaded into reconfigurable hardware under the control of host software. Our contributions are:

- The study of a desktop system in which the CPU triggers FPGA reconfiguration over PCI Express.

– Micro-architectural choices to achieve high transfer rates for reconfiguration and data transactions.
– A transparent process, i.e. without requiring user involvement, for loading and accessing hardware accelerators.

The paper is structured as follows: Section 2 has the related work. Section 3 presents our desktop system combining hardware and software resources. First, we discuss the hardware architecture, and then we detail the software side. At the end of this Section we provide initial results from performance evaluation proving that our approach is promising. Finally, Section 4 concludes the paper.

2 Related Work

Several works exist in the literature studying the use of PCIe for interfacing with an FPGA. The authors of [4] used it for transferring data between the host and the DRAM memory located close to the FPGA. In that design the device operated as a DMA bus master, capable of performing DMA transactions to the main memory and issuing interrupts to the host CPU. The initiation of each transfer was performed by the driver and required that a number of device registers was written. In some of the cases the data had to be fetched from the device DRAM. The design was implemented in a small Virtex 5 at PCIe v1.0 x1 (single-lane) and the total throughput reached, including the device DRAM latency, was 11-15 MB/s. The key bottleneck in that work was the interrupt service rate which was 7680 interrupts/s. They also measured the performance of the DMA from the hardware aspect and they found it equal to 79.3 MB/s for read and 74.1 MB/s for write requests. The device in our case, supports multiple PCIe lanes, leading to higher bandwidth. We also target to providing a complete system capable of reprogramming the FPGA through PCIe.

Another work that is closer to our research is RIFFA [7], in which the authors present a framework for FPGA accelerators dealing with the communication between the accelerators located into the FPGA and the user software. They system is able to manage multiple accelerators implemented in hardware at a given time. They also use the PCIe interface for communication purposes. Our work is significantly different since we currently support a set of resource consuming accelerators running at a given time. We also provide interconnection between the user software and the hardware, but the user in our system is capable of partially reconfiguring the FPGA at a low cost and in a transparent manner.

In a previous work [8], we presented two systems capable of reprogramming the FPGA depending on the user selection. The kernels loaded into the FPGAs were able to communicate with the user application through register I/O. The systems of this work were implemented in two different platforms; the netFPGA and the XUPv5. The key difference between these systems regards their primary interface used for communication purposes. The netFPGA uses the standard PCI interface, whereas the XUPv5 supports PCIe x1. Another important difference is that in netFPGA, a secondary FPGA controls the PCI itself, leading to reconfiguration over PCI. In the present work we are extending the second system,

by enabling DMA operation over PCIe and providing partial reconfiguration capabilities through ICAP.

3 Partially Reconfigurable System

The main contribution of this work is the development of a system able to communicate and partially reconfigure the FPGA, in a manner transparent to the user. Several hardware accelerators can be developed, depending on the user needs, for a certain piece of software, or a software suite. Each of the implemented software packages, may transfer the compute-intensive part of their code into a single or multiple reconfigurable areas of the FPGA. This can be accomplished by a call to our software API containing the ID of the harware module needed, or the partial bitstream itself.

Our aim is to provide a framework capable of programming the FPGA and transferring data in a transparent to the user manner. The user, will be able to implement an accelerator in hardware based on a predefined set of ports and then access it through a software library. Our system will guarantee the user that the accelerator will be executed, but not when the execution will start. Once the execution of an accelerator completes, the system is responsible for reprogramming the PRR with a pending accelerator. Once the reconfiguration is complete, the I/O requests are scheduled and the results are returned to the user. There are several parameters we need to take into account during the implementation of each accelerator and the system itself, both in terms of software and hardware, presented in the following subsections.

3.1 Hardware Architecture

The hardware architecture of our system, illustrated in Figure 1, can be divided in two main regions, the static and the reconfigurable. The static region, contains the PCIe endpoint, the DMA controller and the reconfiguration controller attached to the ICAP port. These components are application independent, as they are responsible for the data transfers and the reconfiguration of the Partially Reconfigurable Regions (PRRs). The reconfigurable space of our system consists of three equally sized PRRs. Each application is located into a PRR and has a set of dedicated memory segments and registers. The total area allocated by the PRRs occupies the 70% of the FPGA in terms of slices, DSPs and routing resources.

The PCIe endpoint of our system is based on the first version of the PCIe interface and it is 4 lanes wide. The PCIe endpoint consists of the physical, the data link, the transaction and the application layers. The data link layer provided as a hardware core [10], while the transaction layer of the PCIe interface in our system was partially based on a hardware application implemented by Xilinx [9]. The PCIe interface itself is packet based, leading to the division of the data transferred to several Transaction Layer Packets (TLPs). The maximum size of each TLP is architecture dependent, while a common selected value in DMA

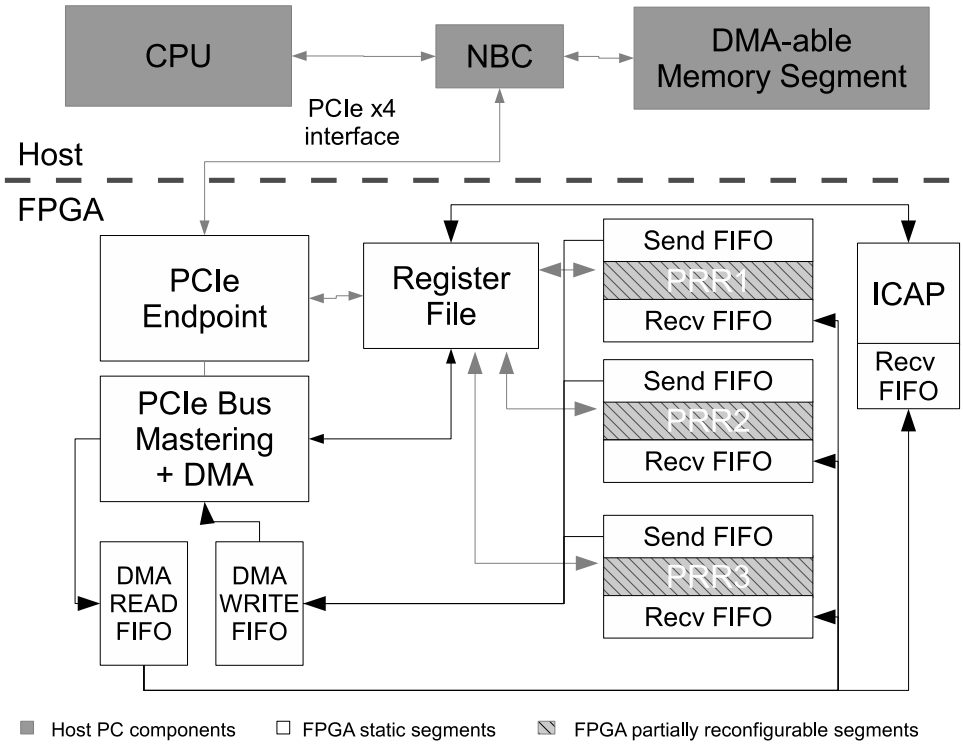

Fig. 1: Hardware Architecture. The hardware components communicate with the host through the PCIe x4 interface. The host operating system initiates DMA transactions between the PRRs, the ICAP and the system memory. The FPGA performs the bus mastering during the transaction.

transactions is 128 bytes. The total number of TLPs transferred is set during the DMA initiation time.

The partial reconfiguration of the FPGA requires a set of steps to be performed until its completion. The first step is the setup of the reconfiguration control register, which is located into the device register file. Once the appropriate word is written to the control register, the reconfiguration controller is waiting for data to be written into its memory block. The bitstream is transferred to the memory block of the reconfiguration controller through DMA. The reconfiguration controller then sends the incoming data to the ICAP, in a sequence of 32 bit words. The consumption of the incoming data leads to an update of the control register value, through which the software is informed to initiate a new DMA. This procedure is repeated until all the reconfiguration data are transmitted, leading to the completion of the reconfiguration process. The access to the DMA buffers is then returned to the application. The status of the configuration procedure is accessible to the operating system through a device register.

The DMA buffers can hold up to 32 KB of data each and are 64 bits wide. Figure 1 depicts only the datapath connections of the data; there are several other associated control signals. There are four 32 bits wide registers, visible to the application for each PRR, three of which are general purpose. The fourth register is holding the ID number of the current bitstream loaded. The other three registers can be used for application specific parameterization and status purposes.

The accelerator itself is the second main component of our system. It resides into one of the three available PRRs, while our system is capable of supporting up to seven PRRs. Each accelerator implements a set of ports for communicating with the rest system. These ports are presented in the Table 1. The system is responsible for sending data to each accelerator and fetching the results. It is informed about the state of each accelerator through a set of events sent via an interrupt. These events are:

- A DMA is complete.
- An accelerator needs data.
- An accelerator has produced results.
- An accelerator has completes its execution.
- The reconfiguration controller needs data.
- The reconfiguration procedure is complete.

clk	input clock operating at 125 MHz
resetN	input active low reset signal
incomingData	input 64 bit incoming data
incomingRen	output read enable for incoming data
incomingEmpty	input incoming data are consumed
outgoingData	output 64 bit outgoing data
outgoingWen	output write enable for outgoing data
outgoingFull	input outgoing data not transmitted
reg1,2	input 32 bit general purpose registers
ressreg	output 32 bit general purpose register
complete	output execution complete
bitstreamID	output 32 bit identification for certain bitstream

Table 1: Accelerator Port Map

In order to cover the case when multiple I/O intensive accelerators are executed, the accelerator has to support a stall state. It needs to enter this state when there are no inputs or there is no room for results. When these issues are resolved by our system, the accelerator has to be able to exit its stall state. Our system needs to be informed in case an accelerator has completed its execution in order to replace it with a new one. The following section presents the software interface provided by our system.

3.2 Software Architecture

The software consists of a device driver and a user space application. The driver, in its current form, is implemented in Linux as a character device driver capable of performing Programmed Input Output (PIO) operations. The memory read or written through these operations, represents the register file of the device. The device operates as bus master, thus it is capable of performing DMA transfers.

Interrupts in our design are used to inform the operating system for the events presented in the previous section. PCIe supports both Message Signal Interrupts (MSI) and legacy interrupts. MSI are issued by performing memory write transactions. The key feature of the MSI is that the total amount of available interrupts is increased. The legacy interrupt emulation is also performed by certain messages. Each legacy interrupt pin (INTA, INTB, INTC and INTD) is represented by two certain messages (Assert and De-assert) led to the system interrupt controller [10].

The device has access to a certain memory segment during the execution of a DMA transaction. This memory segment is allocated in the kernel memory space, at the initialization phase of the driver and its physical address is stored into a device register. The driver has exclusive access to that memory segment, thus an extra copy of incoming and outgoing data needs to be performed. In the user aspect, write and read system calls need to be performed before each DMA read and after each DMA write respectively.

The partial reconfiguration procedure requires a set of operations to be performed. Initially, one of the pending accelerators needs to be scheduled for execution. The bitstream file is located into the data structure holding each accelerator. In order to initialize the reconfiguration procedure, the system needs to set the proper device register and we then transmit sequentially all the bitstream data packets. Finally, a message informs us about the completion of the reconfiguration procedure.

The driver is responsible for keeping and scheduling both the accelerators and the appropriate I/O requests. As a result we need to define a set of function prototypes through which the software will be able to communicate initially with the driver and the accelerator.

The prototypes in Table 2, define an adaptive manner of communication between the software and the hardware. First, the user application needs to create a new accelerator and provide to it the appropriate bitstream files. Then, the application may need to write a set of registers for initialization purposes. After the initialization is complete, the user level software needs to transmit to the accelerator the appropriate data sets. The initiation of DMAs takes place into the driver once the accelerator resides into a PRR. The user level application is then capable of issuing a data receive, which will block until all the requested data are fetched from the FPGA. Finally, once the execution of the accelerator is complete, device is "destroyed".

Generally, the scheduling of each accelerator and the transmission of the required data are completely transparent to the user. The transmission is per-

Accel* CreateDevice ();	Creates an accelerator entry. Returns reference to the accelerator or NULL in failure.
int SetBitsetreams(Accel*, char*, char*, char*);	Assigns the bitstream files for all PRRs. Returns non-zero value in case of an error.
int Transmit(Accel*, void*,int);	Non blocking transmission of data. Takes as parameters the pointer to data and their size.
int Receive(Accel*, void*,int);	Blocking reception of data. Takes as parameters the pointer to data and the size of the data expected.
int ReadDevReg(Accel*,int);	Blocking read of a certain device register.
void WriteDevReg(Accel*,int,int);	Non-blocking write to a certain device register.
void DestroyDevice(Accel*);	Destroys an accelerator entry.

Table 2: Software Prototypes

formed sequentially for all the given data, depending on the needs of the accelerator. As a result, the throughput can be equally shared among the accelerators.

3.3 Evaluation

System	X58 Based
Processor	Core i7 950 @3.0GHz
Chipset	Intel X58 Express
RAM	6GB DDR3 @ 1600
FPGA	Virtex 5 LX330T @ 125MHz
PCIe Interface	Version 1.0, 4 lanes
Reconfiguration Port	ICAP 32bit @ 125MHz
Operating System	CentOs 6.4 @ 64bit
KERNEL Version	2.6.32

Table 3: Desktop System Specifications

Here we are presenting the rates we achieved both for regular data transfers and reconfiguration. Evaluation of hardware components is discussed in this Section, while we are still evaluating of software performance. Table 3 has the system specification. The FPGA operated at 125 MHz. The measurements regard the total throughput reached in terms of time elapsed for transferring 10 GB of data to and from the FPGA. The initialization of each DMA and the reconfiguration procedure are managed by the user-level software, while the driver is character based. We performed DMAs of 32 KB each, since in such transactions the throughput of the interface was saturated. The total throughput reached equal 618 MB/s for DMA Reads (system write) and 544 MB/s for DMA Writes

(system read). The actual measurements are 45-48% lower than the nominal throughput value of the PCIe v1 x4 interface.

As mentioned earlier, the maximum throughput reached for the PCIe interface, is equal to 2.5 GTs per lane. Due to the 10b/8b encoding scheme, it is translated to 250 MB/s per lane. The throughput reached may be considered significantly lower than the theoretical maximum of 250 MB/s per lane (in our case 1 GB/s). That is the rate of byte transfers performed in the physical layer, while there are several bytes added to each TLP among the transaction, the data link and the physical layer of the PCIe. The total amount of extra bytes added, varies between 20-28 depending on the system and device settings, in that way an overhead of 15-21% is added to 128 B TLP transactions [12]. As a result the maximum theoretical throughput, taking into account the packet overheads, is around 200 MB/s per lane(in case there is no packet loss and no acknowledgement costs). In our case, the maximum theoretical throughput would be 800 MB/s and we reached 68% in DMA Writes and 77% in DMA Reads for half duplex transactions.

The partial reconfiguration in our system occurs through writing the incoming bitstream data to ICAP. The interface itself is 32 bit wide and is over-clocked at 125 MHz, giving a throughput of 500 MB/s. The bitstream data are transferred through DMAs and their total size is 1.7 MB per PRR. The reconfiguration throughput reached, is equal to 488 MB/s (97.6% of the maximum). It is important to state that we used double buffering techniques in order to avoid starvation. The partial bitstreams used in this phase, served evaluation purposes. In case we will use custom accelerators, the bitstream size and the overall throughput would remain the same.

4 Conclusions

We presented a desktop system with heterogeneous resources capable to load accelerators without requiring user's involvement. Our work enables execution of multiple applications in reconfigurable resources, or their accessing by multiple users. This can allow to exploit greatly the capabilities of reconfigurable computing, as virtually, it can accommodate an large number of accelerators, even though a certain number of reconfigurable areas is defined at compile time.

We have developed a simple functional interface to ease the use by potential applications, and showed how this functionality is used to transfer the application data between software and hardware by supporting high transfer rates using the PCIe bus. This interface also supports partial reconfiguration through ICAP. Finally, we showed with experimental results that partial reconfiguration can be achieved at high rates.

Acknowledgment

This work was supported by the European Commission in the context of FP7 FASTER project (#287804).

References

[1] Satish K. Dhawan, *Introduction to PCI Express - A New High Speed Serial Data Bus*, IEEE Nuclear Science Symposium Conference Record, 2005.

[2] Matthew J. Koop, Wei Huang, Karthik Gopalakrishnan, Dhabaleswar K. Panda, *Performance Analysis and Evaluation of PCIe 2.0 and Quad-Data Rate InfiniBand*, 16th IEEE Symposium on High Performance Interconnects, 2008.

[3] PCI Sig, *PCI Express 3.0 Frequently Asked Questions*, available at `http://www.pcisig.com`.

[4] Ray Bittner, *Bus Mastering PCI Express In An FPGA*, FPGA'09, Monterey California USA 2009.

[5] Ray Bittner, *Speedy Bus Mastering PCI Express*, FPL'12, Oslo Norway August 2012.

[6] Alan George, Herman Lam, Greg Stitt, *Novo-G: At the Forefront of Scalable Reconfigurable Supercomputing*, IEEE Computer Science and Engineering, Jan-Feb 2011, volume 1 issue 13, pp.82-86.

[7] Matthew Jacobsen, Yoav Freund and Ryan Kastner *RIFFA: A Reusable Integration Framework for FPGA Accelerators*, 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, pp.216-219.

[8] Kyprianos Papadimitriou, Charalampos Vatsolakis and Dionisios Pnevmatikatos *Invited paper: Acceleration of computationally-intensive kernels in the reconfigurable era*, 7th International Workshop an Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012.

[9] XAPP1052, *Bus Master DMA Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions*, Xilinx Corporation, available at `http://www.xilinx.com/support/documentation/application_notes/xapp1052.pdf`.

[10] UG341, *LogiCORE IP Endpoint Block Plus v1.15 for PCI Express User Guide* available at `http://www.xilinx.com/support/documentation/ip_documentation/pcie_blk_plus_ug341.pdf`.

[11] PCI Sig, *PCI Express Base Specification Revision 1.1*, available at `http://www.pcisig.com`.

[12] Alex Goldhammer, John Ayer Jr, *Understanding Performance of PCI Express Systems*, Xilinx WP350, Sept 4, 2008.