A Framework to Improve IP Portability on Reconfigurable Computers

Miaoqing Huang[*], Ivan Gonzalez, Sergio Lopez-Buedo[†], and Tarek El-Ghazawi
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering, The George Washington University
{mqhuang,ivangm,sergiolb,tarek}@gwu.edu

Abstract - *This paper presents a framework that improves the portability and ease-of-use issues of current Reconfigurable Computers (RCs). These two drawbacks should be solved in order for RC to become a mainstream solution. Portability across platforms is difficult to achieve because RC systems have diverse hardware architectures and services. This lack of portability hinders reuse, and thus, ease-of-use. The framework proposed in this work is able to hide the architectural details of the systems, simplify the IP integration, and provide the portability across different RC platforms. User specifies IP requirements such as memory configuration, sequential or random access to the memory, or I/O registers using a graphical-user-interface (GUI) tool, which generates a hardware interface specification for the IP and the logic necessary to target the selected platform. The hardware interface remains the same regardless the targeted architecture. In addition, the tool generates a software library that includes services such as bitstream management and data exchange between microprocessor and IP. This framework has been demonstrated on two representative RCs: Cray XD1 and SGI RASC RC100.*

Keywords: Reconfigurable computing, FPGA, Portable Framework Interface.

1. Introduction

Numerous efforts have proved that Reconfigurable Computers are able to achieve remarkable performance improvements in comparison to traditional microprocessor-based solutions [1, 2, 3]. However, RCs are not reaching a great popularity because reconfigurable hardware devices present two relevant disadvantages when compared to general-purpose processors: ease-of-use and portability.

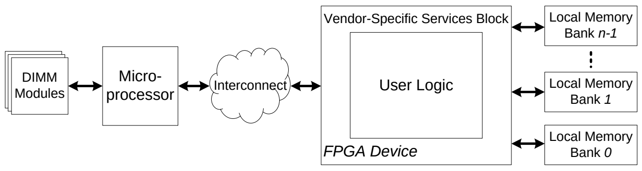


Figure 1. Generic Diagram of Reconfigurable Computer Architecture.

Reconfigurable computers can be defined as general-purpose computers with added FPGA devices working as co-processors [3], see Figure 1. Programming on these systems has two major differences compared to general-purpose microprocessor-based computers. Firstly, generating the FPGA configuration bitstream brings the biggest challenge. The FPGA development flow is completely different from writing software subroutines. New programming languages and tools as well as hardware skills are necessary to accelerate computation applications using FPGA devices. Original algorithms that run on a microprocessor have to be tailored to be able to benefit from FPGA implementation. Secondly, FPGAs are only good for certain categories of computations. Because of this limitation, it is very important to identify the parts of an application that can get significant performance improvement from hardware implementation. Moreover, there is an implicit cost to invoke the FPGA due to the device configuration and communication overhead [4].

Recently, Hardware Description Languages (HDLs) have been challenged by High Level Languages (HLLs) [5, 6, 7] to solve the ease-of-use issue. These HLLs offer a similar syntax as software programming languages such as C. Examples of available HLL languages are Handel-C, Impulse-C, Mitrion-C, Carte-C and so on [8]. However, most of these C-like languages set many restrictions and are not powerful enough to explore the parallelism in hardware and, hence, can not produce highly efficient hardware design for the end users. Furthermore, most of these lan-

[*]This work was supported in part by the I/UCRC Program of the National Science Foundation under the NSF Center for High-Performance Reconfigurable Computing (CHREC).

[†]Visiting from Universidad Autónoma de Madrid.

Table 1. Comparison between Cray XD1 and SGI RC100.

	Cray XD1	**SGI RC100**
FPGA Device	Xilinx Virtex-II Pro-50 FF1152-7	Xilinx Virtex-4 LX200 FF1513-10
Local Memory Architecture	Four 64-bit SRAM modules, 4 MB each	Two 128-bit and one 64-bit SRAM modules, 40 MB in total
I/O (between μP & FPGA)	Bus Width: 64-bit Bandwidth(both directions): 1.6GB/s	Bus Width: 128-bit Bandwidth(both directions): 3.2GB/s
Vendor Services	Hardware perspective: provide means to drive local memory modules and interconnect, through which to access host memory. Software perspective: FPGA is mapped into the IO space of OS and μP is able to issue read and write requests to user logic directly.	Hardware perspective: provide means to access data through register, local memory modules and streaming channels. Software perspective: exchange data between microprocessor and FPGA device automatically.
Execution Model	Interactive: programmer needs to synchronize microprocessor and FPGA device explicitly and manually to perform a task.	Non-interactive: programmer treats the bitstream as software subroutine.

guages do not include support for system integration, therefore, the portability issue between RC systems is not solved by HLLs.

In this paper, we propose a Portable Framework Interface (PFIF) in order to provide the portability and ease-of-use on RC platforms. The remaining text is organized as follows. In Section 2, the motivation for designing PFIF is described. In Section 3, PFIF itself is presented in detail. The experimental result using DES encryption application is demonstrated in Section 4. Finally, we give the conclusion marks and future work in Section 5.

2. Motivation

Since RCs do not share the same hardware architecture model, users have to cope with different FPGA devices, memory configurations, vendor services and execution models when they program on different platforms. Table 1 highlights the main differences between Cray XD1 [9] and SGI RC100 [10] as an example to demonstrate the gaps among different RC systems.

Firstly, RCs have different local memory architectures. The number, size and data width of individual local memory banks are very important for hardware designers and may impact the parallelism of the user logic. Due to the different memory configurations, the supporting logic for data access has to be rewritten when the IP core is ported to a new system. Secondly, vendors generally provide specific services that help access memories and simplify the communication between the microprocessor and hardware cores on their particular platform. These services provide the basic infrastructure to implement data exchange through registers, local memory, host memory, etc. In addition to the service logic for integrating user logic in FPGA device, vendors provide software APIs to configure the FPGA device and to interact with hardware accelerator core. Thirdly, the execution model of FPGA bitstreams varies from one platform to another. On SGI RC100 platform, bitstreams can be considered as software subroutines, i.e., the main program calls the accelerator core and waits until it returns the control back. Alternatively, on Cray XD1, the main program can interactively synchronize with the accelerator core to perform a task. This interactive option provides more flexibility, however, at the cost of a more challenging hardware design.

To summarize, designing hardware-accelerated applications in RC systems is a tedious process. Designers have to study the hardware platform to understand the memory configuration (number of banks, data width and size, etc.), communication interface with the host, vendor services, etc. Then, they maximize the parallelism and performance of the hardware design based on the specification of the target platform. Whenever designers move to a new platform, which has different hardware architecture and vendor services, they have to repeat the same process again.

Wirthlin, Poznanovic and *et al.* from OpenFPGA [11] proposed a solution to address this issue. Basically, they define a standard interface and operating characteristics for designing and integrating IP cores in HLLs and try to push core designers, HLL tool vendors and compiler developers to follow this standard. If RC vendors were able to follow the same specification to design FPGA subsystems and to provide the same services, the portability issue would be perfectly solved. Hence, this solution should be the future trend of RC system development. However, it requires huge effort from both the RC community and vendors to achieve

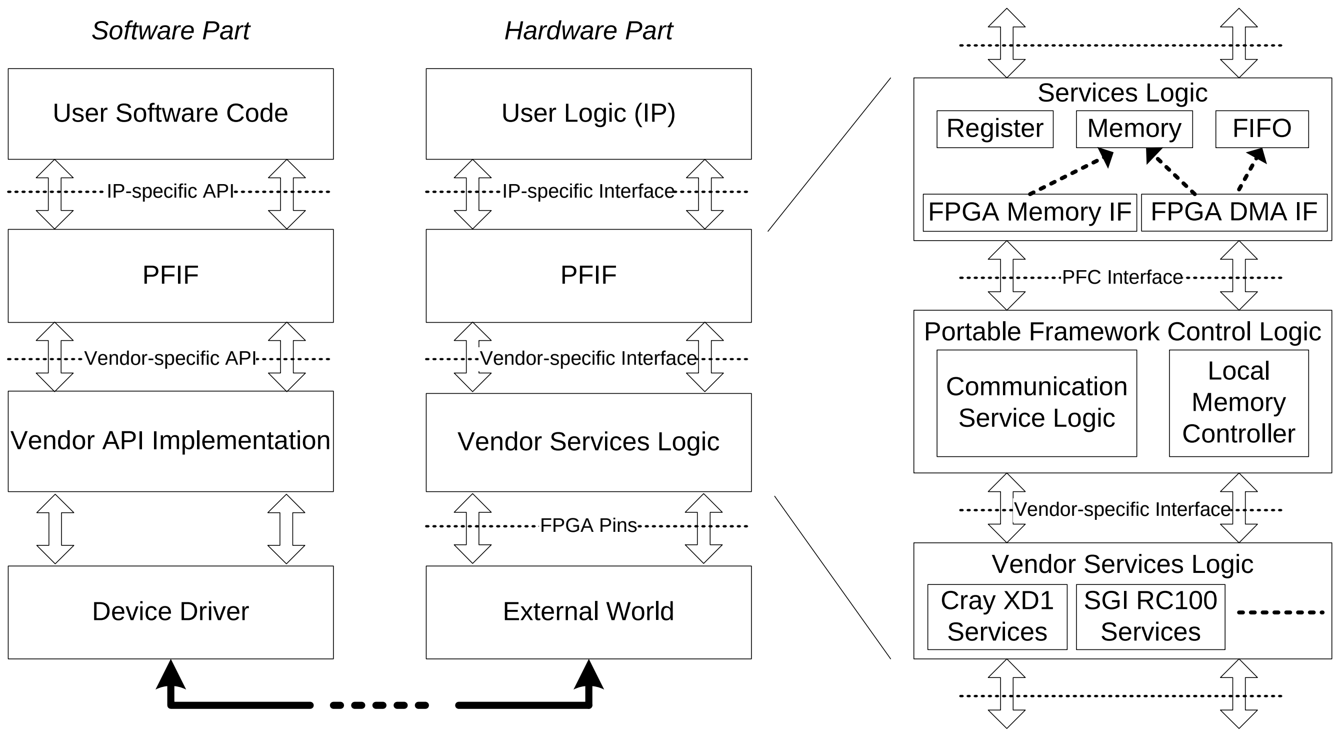


Figure 2. Portable Framework Interface.

this goal.

In this paper, we propose a solution that is more affordable and feasible. The main approach is to hide the difference among the system architectures and services of different RC platforms, and provide a standard layer to reduce the IP integration effort. This standard layer is built on top of vendor's hardware architecture and services, and it is able to generate the user interface (hardware part and software part) based on the application requirements. In the meantime, the design is portable due to the fact that this layer offers the same interface on different platforms even if these platforms do not share the same architecture.

3. PFIF: A Portable Framework Interface

The solution proposed in this work is the Portable Framework Interface (PFIF), which is inspired by Xilinx IPIF approach [12]. Figure 2 shows the PFIF layered architecture and how it is built on top of vendor services and interfaces. Compared to our previous work in [13], which provided a fixed memory access interface across different platforms, PFIF is able to deliver an IP-specific interface on diverse RCs.

3.1. PFIF Overview

At the software side, PFIF provides a set of common functions in order to allow the user to exchange data between the microprocessor and FPGA cores. The IP-specific API is customized to fit the interface configuration used by the IP. These functions will call the specific vendor API, which is provided by the target RC system, to carry out the data exchange. On the hardware side, PFIF appears as a highly parameterizable block between vendor services and the IP. PFIF provides a connection interface that is automatically generated to fit the hardware resources required by the IP. Additionally, PFIF helps the IP development by providing additional services that are ready to use, such as FIFOs, DMA data transfer, registers, etc.

In order to describe the hardware architecture of PFIF, see the right side of Figure 2, we follow a bottom-up approach. The first layer in the PFIF block is called Portable Framework Control Logic (PFCL), which acts as a wrapper of the vendor services and includes the local memory controllers and communication services. This PFCL wrapper is specific for any target system and standardizes the vendor services by means of the Portable Framework Control Interface (PFC Interface). At this point, the portability is achieved because PFC is exactly the same regardless of the target system. PFC is used as the standard interface for

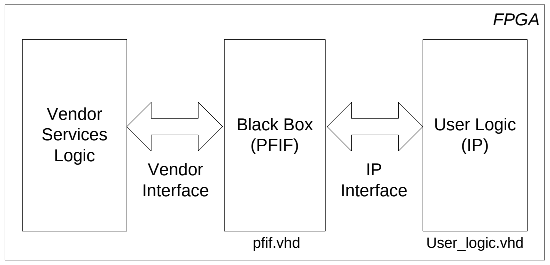
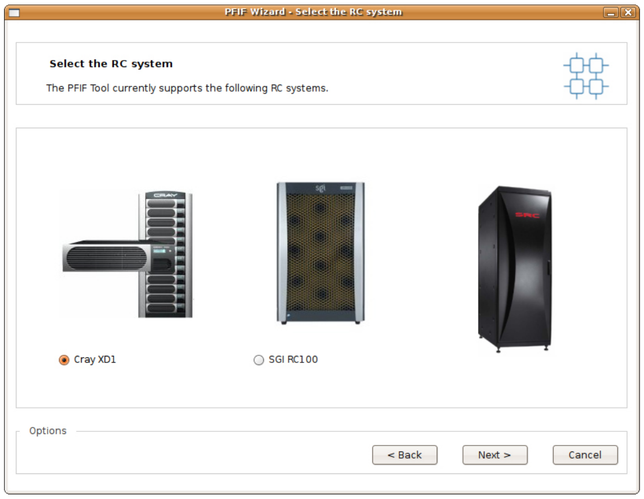


Figure 3. PFIF Design Template.



Select the RC system

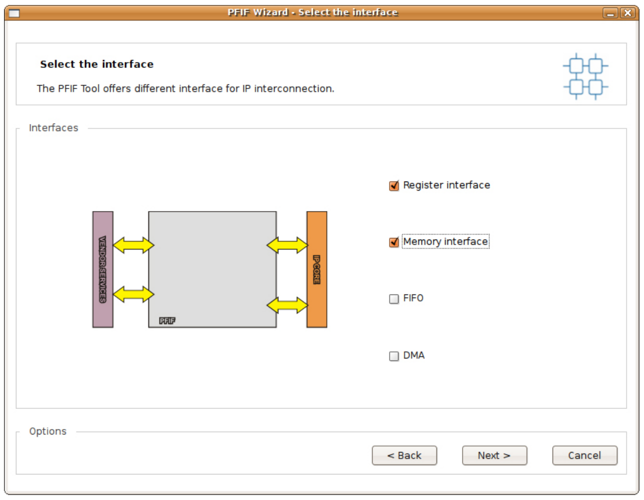
The PFIF Tool currently supports the following RC systems.

[✓ Cray XD1] SGI RC100

Figure 4. PFIF GUI: RC systems currently supported.

the next layer in PFIF, the Services Logic (SL). This layer provides the diverse services that can be used by the IP, including logical memory banks, registers, FIFOs, and also some direct memory access (DMA) capabilities to access the microprocessor memory. Finally, after this layer PFIF provides the IP-specific interface signals for IP integration and design.

The user's point of view of PFIF can be described using a top-down approach, see Figure 3. A GUI tool has been developed to take the input from the user regarding the IP requirements (Figure 4-7): target system, requested services (memory banks, registers, DMA, etc.) and the parameters about these services. After this information is collected, the tool generates a set of HDL files (for hardware integration) and a set of software files (APIs for software integration). The HDL files consist of the user_logic file, including the requested IP interface, and the PFIF file, including the instantiation of the required components from the vendor, the additional components to support the selected services and the control logic to interconnect all these components together to fit the IP requirements.

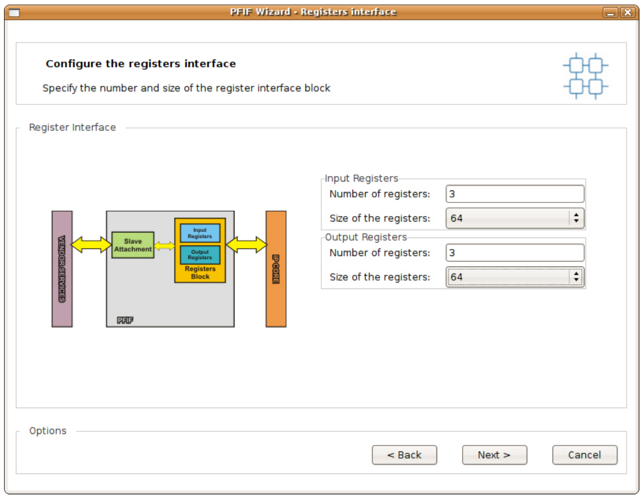


Select the interface

The PFIF Tool offers different interface for IP interconnection.

[✓ Register interface]
[✓ Memory interface]
FIFO
DMA

Figure 5. PFIF GUI: Interfaces.



Configure the registers interface

Specify the number and size of the register interface block

Input Registers

Number of registers: 3

Size of the registers: 64

Output Registers

Number of registers: 3

Size of the registers: 64

Figure 6. PFIF GUI: Register interface.

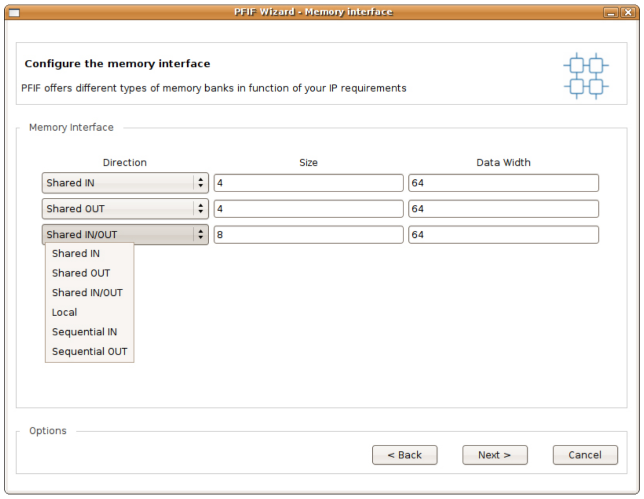
3.2. PFIF Services Interface

The more challenging part of PFIF design is to provide a portable and simple data access interface for user logic. Being portable means the same user logic faces identical interface on different platforms. Being simple means the interface should be configurable, i.e., it should only include the necessary ports for the required functionality.

In general, vendor provides several different techniques for user logic to access data, such as local memory modules, DMA to host memory, and registers. Registers are used for transferring small amount of data, for example, parameters. Local memory modules and direct access to host memory through DMA engines are generally used for transferring huge amount of data in order to optimize throughput. Furthermore, the access to memory contents can be

Table 2. Types of logical memory configurations

Type	Access Direction		Usage	Access Mode (by IP)	Implemented by
	IP	μP			
Shared IN	Read only	Write only	Source data	Random	Local Memory
Shared OUT	Write only	Read only	Result data	Random	Local Memory
Shared IN/OUT	Read & write	Read & write	Source & result data	Random	Local Memory
Local	Read & write	NA	Intermediate data	Random	Local Memory
Sequential IN	Read	Write	Source data	Sequential	DMA
Sequential OUT	Write	Read	Result data	Sequential	DMA


PFIF Wizard - Memory interface

Configure the memory interface

PFIF offers different types of memory banks in function of your IP requirements

Memory Interface

Direction	Size	Data Width
Shared IN	4	64
Shared OUT	4	64
Shared IN/OUT	8	64

Figure 7. PFIF GUI: Memory interface.

sequential or random. In order to make the data access interface portable and simple, these hardware resources are abstracted and user logic can specify the interface based on its characteristics.

The data access interface comprises a set of registers and a set of logical memory banks. Firstly, there are read-only registers for input data and write-only ones for output data. User logic can specify the number of input registers and output registers as necessary, see Figure 6. Because multiple registers can be accessed concurrently, they are good candidates for storing parameters. Secondly, there are two types of logical memory banks, random access memory and sequential access memory, see Figure 7. For random access memory, user can specify the number of access ports, the direction of each port, the data width and the size of each logical memory bank. For sequential access memory, user can specify the access direction (either read or write), and the data width of each bank. For both cases, the user logic only sees the signals of requested access ports. For example, if the user asks for one read-only logical memory bank (the microprocessor will write the content to this logical memory bank), only reading signals are available for the user logic, whereas writing signals are hidden by the services logic.

In addition to the data access interface, PFIF has to generate the logical memory banks requested by the IP using the available local memory modules or the host memory in the selected system. When the user asks for random access memory space, either shared memory space (to exchange data between microprocessor and FPGA) or local memory space (to store intermediate data by the IP), PFIF automatically selects local memory banks. How many local memory modules are sufficient and how they are grouped together to form a logical memory bank is decided based on the data width and size of the requested logical memory banks. For example, if the user asks for one 6MB and 128-bit wide logical memory bank on Cray XD1, two local memory modules are combined. If the user wants one 10MB and 64-bit wide logical memory bank on the same platform, three local memory modules are concatenated together. The data width of the logical memory bank can be less than the physical data width of the local memory module; however, it should avoid generating holes in the physical data storage. For example, the data width of logical memory banks can be 32, 16 or 8, but not 56 or 24. Moreover, user can request data width (of logical memory banks) that is the combination of multiple individual local memory modules, e.g., 128, 192 or 320 bits on SGI RC100. In order to support concurrent access to logical memory banks, a single local memory bank can not be occupied by multiple logical memory banks. For example, if the user asks for two 64-bit wide logical memory banks on Cray XD1, 1MB and 2MB respectively, two local memory banks are used although the combined size of those two logical ones is only 3MB, which is less than the 4MB available in each local memory bank. Another limitation for user is that the total number of local memory banks for implementing logical memory can not surpass the available physical resources. Some of these limitations may be lifted as this research advances, however, the implementation complexity will grow as well. Finally, if sequential access memory is requested, PFIF uses DMA engines inside

Table 3. Resource Utilization and Performance Comparison of DES Encryption on Two Reconfigurable Computers.

Resource Utilization (slices)

Cray XD1		SGI RC100	
W/o Fr	W/ Fr	W/o Fr	W/ Fr
14,645(62%)	14,546(61%)	22,300(25%)	22,300(25%)
End-to-end Throughput (GB/s)			
0.57	0.57	1.14	1.14

the FPGA devices to read and write remote host memory directly. Similar rules of specifying random access memory apply as well. However, in this case, there is no address signal for accessing sequential memory spaces. Reading and writing sequential access memory space always starts from the very first location and in order.

Table 2 summarizes these different types of logical memory banks, their corresponding reading and writing ports with respect to IP core and microprocessor, and other information. PFIF GUI allows the user to choose the desired memory interface among these different types during the PFIF configuration process (see Figure 7).

4. Experimental Result

In order to test the correctness and efficiency of the proposed framework, a DES (Data Encryption Standard) core at ECB mode [14] has been used. The user logic consists of two fully pipelined DES block-ciphers. The parameters of the core, including the encryption key and the amount of plaintext data, are transferred to the user logic through registers. Two DES cores can consume 128-bit data block per clock cycle; hence, two 8MB 128-bit wide logic memory banks are allocated, one for raw data and the other for result data respectively. On Cray XD1, each logical memory bank is implemented by two physical local memory modules. On SGI RC100, every logical memory bank is mapped to one physical local memory module. However, the user logic faces exactly the same interface on both platforms.

Conceptually, the DES encryption application can divided into three steps. Firstly, the plaintext data are transferred from main memory and stored in the local memory of FPGA. After the data transfer is done, the user logic starts reading the raw data from one local memory bank, encrypting them through two DES cores, and writing the result data to another local memory bank in a pipelined way. Finally, the ciphertext data are transferred back to main memory.

```
module user_logic (
   clk, rst, user_logic_go,
   reg_in0, reg_in1,

   mem_0_rd_data,   mem_0_rd_data_vld,
   mem_0_rd_cmd,    mem_0_rd_addr,

   mem_1_wr_cmd,    mem_1_wr_data,
   mem_1_wr_addr,   mem_1_wr_be,

   user_logic_done);

   input            clk, rst, user_logic_go;

   input  [63:0]    reg_in0, reg_in1;

   input  [127:0]   mem_0_rd_data;
   input            mem_0_rd_data_vld;
   output           mem_0_rd_cmd;
   output [18:0]    mem_0_rd_addr;

   output           mem_1_wr_cmd;
   output [127:0]   mem_1_wr_data;
   output [18:0]    mem_1_wr_addr;
   output [15:0]    mem_1_wr_be;

   output           user_logic_done;

   //////////////// Define your user logic after this line ///////////////

endmodule
```

Figure 8. IP Interface Example.

Without the help of PFIF, the user has to deal with these three steps and the synchronization between the software program and hardware logic by himself. If the user designs the application under PFIF, he only needs to focus on the second step, data processing step. PFIF provides the service to automatically exchange data before and after the data processing step, and gives the corresponding signals for the synchronization between PFIF and the user logic. Figure 8 shows the user logic interface in hardware for the DES encryption application. Two registers are allocated for passing the encryption key and the amount of plaintext data. One 8 MB logical memory bank, *mem_0*, is allocated for storing plaintext data. Because the user logic is going to read *mem_0* only, the writing port of *mem_0* is hidden. Correspondingly, the reading port of *mem_1* is not shown to the user either. Another two signals, *user_logic_go* and *user_logic_done*, are provided for the synchronization purpose. After PFIF finishes the raw data transfer from main memory to local memory, it asserts the *user_logic_go* signal; then, the user logic can start functioning. After the user logic completes its processing and writes the last result data block to *mem_1*, it notifies the PFIF by asserting the *user_logic_done* signal so that PFIF can start the result data transfer from local memory to main memory.

The resource utilization and application performance on both platforms are listed in Table 3. The current implementation of portable programming framework does not introduce resources and performance penalty on both platforms due to two reasons. Firstly, although PFIF may introduce some extra latency delay for reading and writing external local memory, the PFIF implementation itself is fully pipelined so that it is able to accept reading or writing request every clock cycle. Secondly, PFIF does the some work as a hardware engineer would do if the data width and depth are different from default ones in the target platform. This is the reason why the resource utilization is the same with and without the framework.

Although the interface signals, such as *mem_x_rd_cmd* and *mem_x_rd_data_vld* for reading data, are same on both systems, the latency may vary when IP core tries to run at the highest possible frequency. For example, on Cray XD1, the default reading latency is 10 clock cycles @ 200 MHz and the default memory data size is 64-bit. In order to implement the 32-bit wide memory, the last digit of read address signal is used for selecting either the upper or the lower 32-bit of a 64-bit word. This arbitration introduces an extra clock cycle delay for reading latency. More latency delay has to be added when a logical memory bank covers multiple local memory banks because a more complex arbitration process is necessary. Although the current implementation of portable programming framework does not introduce resources and performance penalty, as we keep investigating the portability issue and bring more features for improving the flexibility of this framework, costs will eventually appear.

5. Conclusions

A framework to improve the portability and ease-of-use problems on Reconfigurable Computing platforms is presented. The framework provides an abstraction layer to hide the hardware resources of a specific system, simplify the IP integration, and provide portability across different RC systems. Additionally, the framework offers some extra services like registers, logical memory modules, DMA engines, *etc.* to facilitate hardware design. A GUI tool has been developed to generate a particular interface based on user's IP requirements and characteristics. In addition to the hardware design, a software API, which can be used in the software application, is provided for data transfer between microprocessor and IP core.

Future work will focus on HLL support, which makes it possible for HLL IP developers to use the proposed framework. In addition, new efforts will include more sophisticated mechanisms to optimize memory resources, logic resources and performance when the complexity of PFIF increases due to HLL suport and additional features.

References

[1] P. Saha, E. El-Araby, M. Huang, M. Taher, T. El-Ghazawi, C. Shu, K. Gaj, A. Michalski, and D. Buell, "Portable library development for reconfigurable computing systems," in *Proc. The 2007 Reconfigurable Systems Summer Institute (RSSI'07)*, July 2007.

[2] S. R. Alam, P. K. Agarwal, M. C. Smith, J. S. Vetter, and D. Caliga, "Using FPGA devices to accelerate biomolecular simulations," *IEEE Computer*, vol. 40, no. 3, pp. 66–73, Mar. 2007.

[3] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *IEEE Computer*, vol. 41, no. 2, pp. 78–85, Feb. 2008.

[4] O. D. Fidanci, D. Poznanovic, K. Gaj, T. El-Ghazawi, and N. Alexandridis, "Performance and overhead in a hybrid reconfigurable computer," in *Proc. 10th Reconfigurable Architectures Workshop (RAW 2003)*, Apr. 2003, pp. 176–183.

[5] Z. Guo and W. Najjar, "A compiler intermediate representation for reconfigurable fabrics," in *Proc. International Conference on Field Programmable Logic and Applications, 2006 (FPL 2006)*, Aug. 2006, pp. 741–744.

[6] J. L. Tripp, M. B. Gokhale, and K. D. Peterson, "Trident: From high-level language to hardware circuitry," *IEEE Computer*, vol. 40, no. 3, pp. 28–37, Mar. 2007.

[7] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, "DWARV: Delftworkbench automated reconfigurable VHDL generator," in *Proc. International Conference on Field Programmable Logic and Applications, 2007 (FPL 2007)*, Aug. 2007, pp. 697–701.

[8] E. El-Araby, M. Taher, M. Abouellail, T. El-Ghazawi, and G. Newby, "Comparative analysis of high level programming for reconfigurable computers: Methodology and empirical study," in *Proc. IEEE 3rd Southern Conference on Programmable Logic 2007 (SPL 2007)*, Feb. 2007, pp. 99–106.

[9] *Cray XD1^{TM}FPGA Development (S-6400-14)*, Cray Inc., May 2006.

[10] *Reconfigurable Application-Specific Computing User's Guide (007-4718-006)*, Silicon Graphics, Inc., Mar. 2007.

[11] M. Wirthlin, D. Poznanovic, and *et al.*, "OpenFPGA corelib core library interoperability effort," in *Proc. The 2007 Reconfigurable Systems Summer Institute (RSSI'07)*, July 2007.

[12] *Xilinx OPB IPIF (v3.01c) Data Sheet*, Xilinx, Inc., Dec. 2005.

[13] M. Huang, I. Gonzalez, and T. El-Ghazawi, "A portable memory access framework on reconfigurable computers," in *Proc. IEEE International Conference on Field-Programmable Technology 2007 (ICFPT'07)*, Dec. 2007, pp. 333–336.

[14] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. Boca Raton, FL: CRC Press, Dec. 1996.