A Portable Memory Access Framework on Reconfigurable Computers

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

Abstract

Current Reconfigurable Computers (RCs) do not share a unified architectural model, which presents a challenge to any developer who intends to port hardware designs across different RC platforms. In this paper, we propose a portable memory access framework that gives the user a unified memory view combining the host memory and the local memory of FPGA. Three memory access modes are provided, and the hardware cost and performance impact have been measured on three major RCs: SRC-6, SGI RC-100 and Cray XD1. Under current implementation, the penalty to hardware resource utilization and performance of applying this framework is reduced to minimum.

1. Introduction

In the past few years, several vendors have introduced new systems that contain both microprocessors and FPGAs such as SRC-6, SGI Altix/RASC systems, and Cray XD1. These systems can be considered parallel computers that resemble modern HPC architectures, with added FPGA devices working as co-processors to the microprocessor(s). In these architectures, the main application will be executing on the microprocessor(s) where kernels that take long execution time, but lend themselves to hardware implementations are extracted on the FPGA.

Although several High Level Languages (HLLs) targeting RCs are available currently, Hardware Description Languages (HDLs) are still the primary means for exploiting the parallelism of FPGA devices. However, integrating a processing core into a new platform is not a trivial task, especially for HDL developers. RCs have different local memory architectures, network fabrics, and development frameworks, which requires the developer to spend tremendous effort for integrating processing core into FPGA in addition to the effort for developing the core itself.

To overcome this difficulty, a standard memory access framework on RCs is proposed. We consider following features to make this framework portable and easy to use: (1) The memory framework unifies the host memory and the local memory into a logical memory space. (2) The user logic only deals with the logical memory space for data access. The framework takes care of the physical data transaction between FPGA and μP. (3) The framework maximizes the overall data throughput under various circumstances. In the following description, we dedicate the term "host memory" to the main memory managed by software operating system and the term "local memory" to the memory modules connected to FPGA device.

Several related works have discussed the integration of μPs and FPGAs, and different solutions have been proposed. For example, recently, Vuletić *et al.* [1] introduced a system layer with hardware support to bring the local memory of reconfigurable hardware resource (FPGA or ASIC) into the virtual memory space which is managed by the OS. Similar approaches have been considered in [2, 3], where the OS is used to manage FPGA resources. Other research works have tried to solve the communication interface problem. For example, Maalej *et al.* [4] described a methodology to design a communication interface that directly connects the hardware module to general-purpose processor for System On Chip system.

However, these previous works focus more on software system level on generic reconfigurable platforms and then are different from the approach presented in the paper. The main differences comprise: (1) The target platforms of the proposed framework are fully developed RCs such as SGI RASC and Cray XD1. On these platforms, vendors provide a platform-specific interface block (as shown in Figure 1), software drivers and APIs for the communication and synchronization between μP and FPGA. (2) The target end users are hardware designers who intend to integrate their processing cores into vendor-specific programming environment using HDL.

*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).

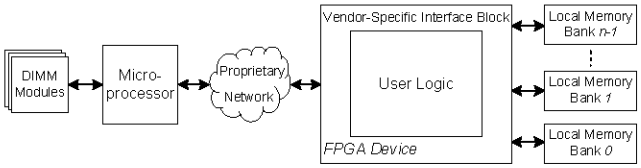

Figure 1. Generic Architecture of Reconfigurable Computer.

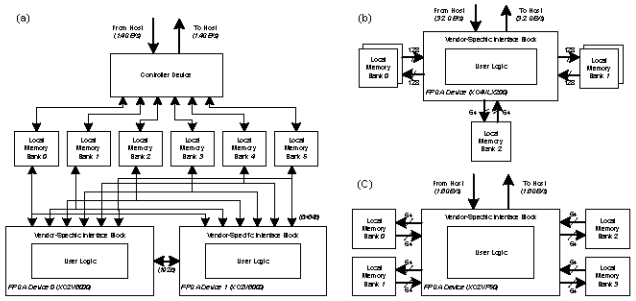

Figure 2. Local Memory Architectures of three major RCs: (a) SRC-6, (b) SGI RC-100, (c) Cray XD1.

The portable memory access framework is built on the top of the vendor-specific interface block, and the memory access interface offered to the user logic keeps unchanged across different platforms. In order to realize this objective, two sets of files are provided to hide all the details behind the framework interface. One set of HDL files will give the end user the hardware interface of the logical memory. The other set of HLL files will specify the behavior of the μP according to the bitstream.

2. Local Memory Architecture and Access Methods of Representative RCs

Figure 1 shows the generic view of the FPGA device and its local memory architecture, which consists of several independent local memory banks. Generally, the user logic is surrounded by the vendor-specific interface block, which provides the user logic the means to access the outside world.

On SRC-6 platform, two FPGA devices share 6 banks of local memory, 24MB in total,as shown in Figure 2(a). Because the user logic only has one port for reading and writing of local memory, it can not perform both transactions on same bank simultaneously. There are two strategies for handling data transportation between local memory and host memory on SRC-6. (1) **non-overlapping:** There is no overlap among data transferring-in, data processing and data transferring-out. (2) **semi-overlapping:** Either data transferring-in and data processing are overlapped, or data processing and data transferring-out are carried out concurrently, however not both.

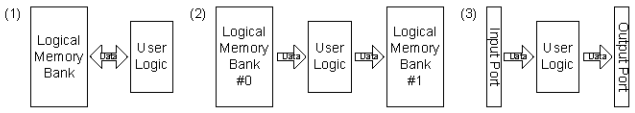

Figure 3. Three Memory Access Modes: (1) Dual-ported random access mode, (2) Single-ported random access mode, (3) Sequential access mode.

SGI's latest RASC technology, RC-100, has 5 banks of physical SRAM, 40MB in total. However, the user logic only views three local memory banks, as shown in Figure 2(b). The user logic has separate reading and writing ports with local memory and can carry out both operations on same memory bank concurrently. On RC-100, the user must use one of following three strategies to transfer data. (1) **non-overlapping:** Same to corresponding case of SRC-6. (2) **multi-buffering:** Overlap data transferring-in, data processing and data transferring-out. Two memory banks are dedicated for raw data and result data respectively. Both memory banks are divided into multiple windows, therefore vendor logic and user logic can access the same bank simultaneously, however, on different windows and performing disparate transactions. (3) **streaming DMA:** Raw data are transferred into user logic and result data are returned back to host memory directly.

On Cray XD1 platform, one FPGA device is connected with four banks of local memory, 4MB each, as shown in Figure 2(c). Same to RC-100, each local memory bank has two separate ports for reading and writing respectively by user logic. Different from previous two cases, Cray XD1 does not pre-define any data transportation strategy and the host program has to synchronize with the bitstream interactively to process a task. Therefore the user has to work out his own strategy to fit his application.

3. The Proposed Memory Access Framework

In order to minimize the loading of the hardware developers to handle data movement between host memory and local memory and make their designs portable, the framework provides a single logical memory space that unifies both memories. Furthermore, in order to make the memory access framework as efficient as possible, three different access modes, as shown in Figure 3, are defined in the remaining part of this section.

In the dual-ported random access mode, the framework

Table 1. The interface signals defined in memory access framework.

Signals	I/O	Description
user_logic_go	I	If asserted, user logic can start function
mem_rd_rq	O	Assert this signal if *mem_rd_addr* valid
mem_rd_addr [19:0]†	O	Specify the reading address
mem_rd_vol [31:0]‡	I	Specify the data volume to be read
mem_rd_data_vld	I	If asserted, *mem_rd_data* is valid
mem_rd_data [127:0]	I	Valid if *mem_rd_data_vld* asserted
mem_wr_rq	O	Assert this signal when *mem_wr_data* and *mem_wr_addr* are both valid
mem_wr_addr [19:0]†	O	Specify the writting address
mem_wr_ready‡	I	If asserted, writing is allowed
mem_wr_data [127:0]	O	Output data
user_logic_done	O	Assert this signal if user logic is done

**In the above case we assume the data width is 128-bit and the size of one logical memory bank is 16MB.*

†: *random access mode only.* ‡: *sequential access mode only.*

provides one logical memory bank and the user logic can access any memory location randomly, as shown in Figure 3(1). A typical application requiring this memory access mode is large-size data sorting in which data are processed through multiple rounds to reach the desired order.

Signals for handling this mode are shown in Table 1. After the framework finishes preparation, it asserts the signal *user_logic_go* to let the user logic start. Similarly, the user logic should assert the signal *user_logic_done* to notify the framework the end of its operation and then the framework will make sure all result data are written back to host memory before terminating the bitstream. The usage of these two signals are same in three modes. In mode-1, when the user logic reads the memory, the data blocks are delivered as the same order as requested and the validity is indicated by the signal *mem_rd_data_vld*. For memory writing, the address and data are required to be given at the same time.

In the single-ported random access mode, the framework provides two separate logical memory banks for storing raw data and result data respectively, as shown in Figure 3(2). The interface signals in this mode is same to the previous one. However, the reading and writing directions of user logic never change during the whole period of hardware process, which is the functionality of the bitstream.

The sequential access mode supports the computation scenario in which all the data access is in order. Only two ports are required for dealing this situation, one for reading and the other for writing, as shown in Figure 3(3). A typical example of this scenario is data encryption and decryption under Electronic Codebook (ECB) mode in which the message is split into blocks and each block is processed separately [5]. The signal *mem_access_vol* tells the user logic how many raw data blocks are to expect. The user logic has to check the signal *mem_wr_ready* before it tries to write result data to output port.

4. Implementation

Because the three platforms have different local memory architectures and data transferring mechanisms, the implementation of this memory access framework varies from one machine to another. Table 2 lists the mapping between the three modes and the three platforms. In this initial implementation, the data width is set to 128-bit wide for all three modes. The memory size of one logical bank is fixed as 16 MB and 8 MB for mode-1 and mode-2 respectively.

The user logic for testing these three modes consists of two fully pipelined DES (Data Encryption Standard) block-ciphers [5]. Running at 200 MHz, these two cores can produce a theoretical throughput of 3.2 GB/s (16B×200M/s). However, the real performance largely depends on the means to transfer the data and the overlap between data transferring and data processing.

For mode-1, because the user logic takes control of both reading port and writing port of the same memory bank during its function period, there is no overlap between data transferring and data processing.

Mode-2 is implemented by using the "Multi-buffering" mechanism on SGI RC-100. Logical memory banks 0&1 are mapped to half of local memory banks 0&1 respectively. Both local memory banks 0&1 are divided into two windows such that vendor logic and user logic can access same local memory bank simultaneously to overlap the data transferring and data processing. On other two platforms there is no overlap because there are no enough spare local memory banks for implementing similiar mechanism.

For mode-3, the data transferring-in and data processing are overlapped on SRC-6. On other two platforms, the local memory banks are completely bypassed. The source data blocks are read from host memory and fed to user logic directly. The result data blocks are written back to host memory without local store as well.

For mode-1 & 2, if the size of raw data is bigger than the size of the logical bank, multiple iterations of hardware process will be triggered. The framework will take care of the data transferring automatically and start the user logic several rounds to perform the data processing.

Table 3 shows the resource utilization and performance of both cases, with or without the framework, of these three different modes. The proposed framework specification introduces negligible resource penalty in the hardware design

Table 2. Mapping three memory access modes onto three reconfigurable computers.

	SRC-6	SGI RASC*	Cray XD1
mode-1 Fig.3(1)	Four local memory banks are combined as one logical bank.	Whole local memory bank 0 is treated as one logical bank.	Four local memory banks are combined as one logical bank.
	No overlap between data transferring and data processing.		
mode-2 Fig.3(2)	Two local memory banks are combined as logical bank 0, and another two local memory banks are combined as logical bank 1. No overlap between data transferring and data processing.	Half of local memory bank 0 is treated as logical bank 0. Half of local memory bank 1 is treated as logical bank 1. Use the "Multi-buffering" mechanism to overlap data transferring and data processing.	Two local memory banks are combined as logical bank 0, and the other two local memory banks are combined as logical bank 1. No overlap between data transferring and data processing.
mode-3 Fig.3(3)	Two local memory banks are combined to provide the raw data, and another two local memory banks are combined to store the processed data. Data transferring-in and data processing are overlapped. Data transferring-out is carried out after the data processing.	Use the "Streaming DMA" mechanism to retrieve raw data from host memory and return result to host memory directly.	"Host memory direct access" mechanism is adopted. The user logic reads from and writes to host memory through the "User Request Interface" of the "RapidArray Transport Core".

**On SGI RASC platform, allocate "hugepage" to avoid the kernel copy at the host side.*

Table 3. Resource Utilization and Performance Comparison of Three Memory Access Modes on Three Reconfigurable Computers.

Resource Utilization (slices)

	SRC-6	SGI RC-100		Cray XD1	
		W/o Fr	W/ Fr	W/o Fr	W/ Fr
m-1	14,317	22,281	22,281	14,942	14,685
m-2	14,172	22,300	22,300	14,645	14,546
m-3	15,007	22,659	22,659	13,670	14,428
End-to-end Throughput (GB/s)					
m-1	0.02	0.65	0.65	0.57	0.57
m-2	0.47	1.14	1.14	0.57	0.57
m-3	0.67	2.09	2.09	1.17	1.17

and does not alter the final performance regardless of the the memory access mode.

5. Conclusion and Future Work

A portable memory access framework on reconfigurable computers is proposed. This framework is built on the top of vendor's pre-defined interface block and provides a unified logical memory space that combines the local memory and the host memory. Three memory access modes, Dual-ported Random Access Mode, Single-ported Random Access Mode and Sequential Access Mode are provided for different scenarios. Preliminary testing of this framework on three major RCs shows no resource utilization overhead and performance impact. However, one major limitation of the current implementation is the lack of flexibility. The quantity, size and data width of memory banks are all fixed, which undermines ease-of-use and coverage of this framework. A tool is under developing to customize these three characteristics of the memory interface following user's request. In addition to increasing the ease-of-use, the tool will try to reduce the resource cost and performance impact to minimum as well.

References

[1] M. Vuletić, P. Ienne, C. Claus, and W. Stechele, "Multithreaded virtual-memory-enabled reconfigurable hardware accelerators," in *Proc. IEEE International Conference on Field Programmable Technology 2006 (FPT 2006)*, Dec. 2006, pp. 197–204.

[2] M. Dales, "Managing a reconfigurable processor in a general purpose workstation environment," in *Proc. Design, Automation and Test in Europe Conference and Exhibition, 2003 (DATE'03)*, Mar. 2003, pp. 980–985.

[3] M. Vuletić, L. Pozzi, and P. Ienne, "Virtual memory window for application-specific reconfigurable coprocessors," *IEEE Trans. VLSI Syst.*, vol. 14, no. 8, pp. 910–915, Aug. 2006.

[4] I. Maalej, G. Gogniat, M. Abid, and J. L. Philippe, "Interface design approach for system on chip based on configuration," in *Proc. 2003 International Symposium on Circuits and Systems*, vol. 5, May 2003, pp. 593–596.

[5] M. Huang, T. El-Ghazawi, B. Larson, and K. Gaj, "Development of block-cipher library for reconfigurable computers," in *Proc. IEEE 3rd Southern Conference on Programmable Logic 2007 (SPL'07)*, Feb. 2007, pp. 191–194.