Improving the Performance of Shared Memory Communication in Impulse C

X. Jin, N. Guan, Mingsong Lv and Qingxu Deng

Abstract—With the evolution of Field Programmable Gate Arrays (FPGAs) to the Million-Gate scope, high-level languages are gaining popularity in electronic system design, which greatly improves design and verification efficiency. Impulse C is a high-level language widely used in SW/HW co-design and provides users with varies SW/HW communication mechanisms. But the communication mechanisms of Impulse C are mainly designed for versatility, and the resources within the FPGA chip is not fully utilized. In this article, we present a improved implementation of the shared memory communication in Impulse C by utilizing both ports of the dual-port BRAM. Experiment results show that the improved implementation can greatly improve the performance of shared memory communication, and further improve the execution efficiency of hardware processes.

Index Terms—Shared memory, SW/HW communication, Impulse C

I. INTRODUCTION

With the development of deep sub-micron technology, millions of gates can be integrated on a single FPGA chip, the design of large scale application systems using FPGA becomes possible. Flexible software modules and high-performance hardware modules are usually combined to implement sophisticated high-performance embedded systems [1], [2]. Traditionally, FPGA-based hardware modules are designed either by hardware description languages, such as VHDL and Verilog HDL, or by GUI-based approaches in which function units are specified by functional blocks. These design methodologies have two major problems. First, system designers are intensively involved in the design process and proficiency in hardware description languages is mandatory, so the methodologies cannot scale to the design of complex application systems that typically utilize millions of gates. Second, traditional design methodologies are hard to meet the needs of SW/HW co-design and co-verification.

With this background, high-level languages, such as Handel-C [3], FpgAc [4], SPARK [5], ROCCC [6], Catapult C [7], and Impulse C [8], are gaining popularity in FPGA-based electronic system design. Such high-level languages allow system designers to model a system at a higher abstraction level, which greatly improves design efficiency. Impulse C is one of the most widely used languages, since it has large coverage of the ANSI C standard and provides users with a convenient GUI development environment. Impulse C also provides libraries specific to different FPGA platforms, which further improves design efficiency. Most of all, among all the above mentioned languages, only Impulse C allows mapping an application to a platform with both microprocessors and FPGA, which simplifies HW/SW co-design. So we focus our discussion on Impulse C in this article.

CoDeveloper is the integrated development environment that supports Impulse C and SW/HW co-design. The basic program unit of Impulse C is process, and an application is typically composed of parallel software and hardware processes that communicate via the inter-process communication mechanisms provided by Impulse C. Impulse C has three major communication mechanisms: (1) signals, (2) data stream and (3) shared memory. Signal is a unidirectional communication mechanism for small volume data transfer, which is usually used in synchronizing processes and external events. Data stream is a unidirectional synchronous communication channel between HW/SW processes. An application may read or write multiple data streams. If the stream is full or empty, hardware processes are blocked, while software processes poll the stream until the data are ready. Different from data stream, shared memory allows communicating processes to read or write shared memory independently, and synchronization is only specified by application developers if required. So it has better performance in transferring large volume data. Shared memory also allows random data access, so the processes can manipulate data in a more flexible manner. In this article, we focus our discussion on shared memory based communication.

On Xilinx platforms, shared memory can be either internal dual-port BRAM [9] or external memory. In the original Impulse C implementation, processes can only access the shared memory via OPB (On-chip Peripheral Bus) and the OPB memory interface controller. This access method can guarantee design versatility, but the communication performance is limited by bus access overhead. In this article, we present a improved implementation of shared memory communication which bypasses system buses and enables user logic to directly access the BRAM by utilizing the other unoccupied BRAM port, thus the performance of shared memory communication and the execution efficiency of hardware processes are greatly improved. We designed a new interface controller called SMCI (Shared Memory Communication Interface) as well. The module is integrated into the Impulse C architecture and is completely transparent to system designers.

The rest of this article is organized as follows. Section II presents the improved communication mechanism and details
the implementation. Section III gives the experimental results, and the article is concluded in Section IV.

II. THE IMPROVED SHARED MEMORY COMMUNICATION

In this section, we first present the original hardware architecture of the shared memory communication of Impulse C. Then the modification to the hardware architecture of the original shared memory communication mechanism is introduced. The implementation of SMCI and the integration of SMCI to the CoDeveloper development environment is detailed as well.

A. The Original Hardware Architecture and Its Limitation

The original hardware architecture of the Impulse C shared memory communication mechanism is illustrated in Fig. 1. External memory and internal BRAM are all shared memory, and Impulse C requires that all memory modules must be connected to OPB. If a software process that runs on the PowerPC core tries to access the shared memory, it has to access PLB (Processor Local Bus), OPB and "Opb_BRAM_if_cntlr" sequentially (the dashed line in Fig. 1). Hardware processes are custom IP Cores represented as PLB slave and OPB master. A hardware process must use the shared memory access controller "Opb_dma" to access shared memory via OPB and "Opb_BRAM_if_cntlr" (the dotted line in Fig. 1). The "Communication Interface" is the connection between PLB and user logic. Two unidirectional communication channels that implement the Impulse C signal mechanism are designed to make a duplex channel to synchronize software processes and hardware processes. Hardware processes get the base address of shared memory via data stream.

Standard bus interfaces, such as "OPB_BRAM_if_cntlr", allow software or hardware processes to access any type of memory units connected to OPB. In this way, only one port of BRAM is connected to "OPB_BRAM_if_cntlr", which limits the bandwidth of BRAM access. While nearly all Xilinx FPGA chips are equipped with dual-port BRAM, if we can enable hardware processes to access BRAM directly, then the R/W data rate can be greatly improved. For instance, if the BRAM has 32-bit data width and works at 100MHz, ideally a 400MB/s bandwidth could be reached. This is the basic idea of improving shared memory communication performance in this article. Even if the design is implemented on the PLB bus alone, accessing BRAM via PLB introduces bus overhead, too. In this case, our approach is still effective since HW processes can access BRAM via the direct link.

B. The Improved Hardware Architecture

Fig. 2 illustrates the improved hardware architecture of the shared memory communication mechanism. We designed a new intermediate interface controller between "User logic" and "Opb_dma", called "SMCI". The first function of this module is path selection. If a hardware process tries to access shared memory, the request is send to SMCI. Then SMCI checks whether the hardware process is accessing BRAM by identifying the range of memory address specified by C_SBRAM_BASEADDR and C_SBRAM_HIGHADDR. If the hardware process is to access BRAM, then there is no need to access OPB since a direct data path between SMCI and BRAM is established. If the hardware process is to access shared memory other than BRAM, it will use the Opb_dma interface controller to access shared memory via OPB, which is identical to the original mechanism. The second function of SMCI is interface control that manages the timing conversion between the shared memory access interface of "User logic" and the BRAM interface.

C. Implementation of the SMCI

Fig. 3 illustrates the detailed implementation of SMCI. SMCI has two modules: the path selection module and the BRAM interface control module.

When using BRAM as the shared memory, the communication path between User logic and BRAM is enabled. There are two operation modes of shared memory access in Impulse C. The block mode allows hardware processes to read or write block data via co_memory_readblock() and co_memory_writeblock(). While the single mode allows a hardware process to access single memory address which is assigned to a specific pointer of the hardware process via co_memory_ptr(). The hardware process then accesses the memory address using this pointer. The read and write under both operation modes are supported in our improved
architecture. In Fig. 3, the comp_block signal is used to control the operation mode of SMCI, and the comp_mode signal is used to control read and write transfers.

When a hardware process accesses the shared memory, it first enables the comp_req signal, and then configure the number of bytes to access, the type of the data and the memory base address by setting the comp_count, comp_size and comp_base signals, respectively. Note that the comp_count signal is not used in single mode. If a hardware process writes the shared memory, the comp_wri signal controls the loading of the data at the address specified by comp_nextaddr to the comp_odata port; then the bram_wen signal controls the data copying from the comp_odata port to the bram_din port and writes this data to the address specified by bram_addr. Since the bit width of the data might be different, so data type conversion is required in the data copying process. The reading of the shared memory is similar, and the only difference is that data is copied from the bram_dout port to the comp_idata port. When the read or write operation is finished, SMCI will send a signal to the hardware process via the comp_ack signal to notify data transfer completion.

When a hardware process accesses shared memory other than BRAM, the path from user logic to Opb_dma is enabled, and SMCI simply performs data forwarding.

D. Integration of SMCI with CoDeveloper

An Impulse C hardware process is compiled as three VHDL files by CoDeveloper. Fig. 4 illustrates the relation between the files. The *_comp.vhd file specifies the functionality of a user logic. The *_top.vhd file specifies communication entity and encapsulates the user logic and the communication entity, and also implements the bus access interface (impulse_plb_ipif, communication bridge and opb_dma). SMCI is implemented in the plb_*_arch.vhd file, so the *_top.vhd file and the *_comp.vhd file need not to be modified.

The SMCI is integrated within the CoDeveloper environment. Fig. 5 illustrates the context of SMCI in the development environment. The modules within the dashed rectangle is the original CoDeveloper modules, and SMCI is inserted into the plb_*_arch.vhd file after the "Generate host/FPGA interfaces" (depicted as shadowed area). When building systems with the improved architecture, the connection between SMCI and the BRAM port should be established, and the assigned BRAM address range should be configured by setting C_SBRAM_BASEADDR and C_SBRAM_HIGHADDR. The VHDL code of SMCI is completely implemented in the plb_*_arch.vhd file and outside the *_top.vhd file. The original cross-file interfaces and the design flow of CoDeveloper are not changed, so the application code needs no modification.

III. EXPERIMENTAL RESULTS

The experiments of this article are based on Xilinx XUPVIIP [10] development board, and the software environment is ISE 9.1, EDK 9.1 and Codeveloper 2.20. First, we compared the read/write performance under different operation modes between the original and the improved architecture of the shared memory mechanism. Fig. 6 and Fig. 7 illustrate the access time ratio of improved architecture to the original one in transferring different sizes of data blocks. Since hardware
processes need not access the BRAM via OPB in our design, the access performance is improved greatly. The average access time ratio is 30% and 44% under "block" operation mode and "single" operation mode, respectively. The results show that the performance improvement is stable regardless of the data volume to be transferred.

![Fig. 6. The access time ratio of the improved architecture to the original architecture under "block" operation mode](image6)

We selected 6 benchmark programs and implemented them as hardware processes to observe the performance gain as well. Table I lists the number of cycles that each benchmark program executes with and without the SMCI interface controller. An average of 29.25% performance gain is witnessed. Since the number of clock cycles includes both shared memory access time and hardware logic execution time, if a program has more shared memory access operations, then the performance improvement is bigger.

We also compared the resource usage after adding the SMCI. Table II lists the number of slices with and without the SMCI. The number of slices listed includes the slice for the corresponding hardware process and the basic modules such as PLB, OPB, bridge and UART. Experiment results show that adding SMCI introduces an average of 4.1% resource overhead, which is reasonable in FPGA based system design. Since adding SMCI does not affect the usage of BRAM and Mult18X18, we omitted the usage of these two resources.

![Fig. 7. The access time ratio of the improved architecture to the original architecture under "single" operation mode](image7)

**IV. CONCLUSION AND FUTURE WORK**

In this article, we present a improved implementation of the shared memory communication in Impulse C by utilizing the dual ports of BRAM. A new shared memory interface controller is designed and integrated in the CoDeveloper environment, which is transparent to application designers. Experimental results on the Xilinx PowerPC platform show that the shared memory access performance and the execution efficiency of hardware processes are greatly improved, which only introduces very small resource overhead.

With the improved communication performance, the results of HW/SW partitioning may be different (e.g. more components could be implemented as HW since there is more memory access bandwidth available). We would like to study the interplay between communication performance and HW/SW partitioning in our future work.

**TABLE I**

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Original Architecture</th>
<th>Improved Architecture</th>
<th>Performance Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR</td>
<td>25,174</td>
<td>22,398</td>
<td>11.0%</td>
</tr>
<tr>
<td>FFT</td>
<td>2,549</td>
<td>1,359</td>
<td>48.7%</td>
</tr>
<tr>
<td>3DES</td>
<td>51,298</td>
<td>41,086</td>
<td>19.9%</td>
</tr>
<tr>
<td>Sobel</td>
<td>1,370,229</td>
<td>1,042,548</td>
<td>23.9%</td>
</tr>
<tr>
<td>Roberts</td>
<td>1,110,059</td>
<td>782,418</td>
<td>29.5%</td>
</tr>
<tr>
<td>CRC</td>
<td>14,452</td>
<td>8,311</td>
<td>42.5%</td>
</tr>
</tbody>
</table>

Average Improvement: 29.25%

**TABLE II**

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Original Architecture</th>
<th>Improved Architecture</th>
<th>Resource Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR</td>
<td>2,030</td>
<td>2,100</td>
<td>3.4%</td>
</tr>
<tr>
<td>FFT</td>
<td>3,627</td>
<td>3,699</td>
<td>2.0%</td>
</tr>
<tr>
<td>3DES</td>
<td>2,884</td>
<td>2,934</td>
<td>1.7%</td>
</tr>
<tr>
<td>Sobel</td>
<td>2,016</td>
<td>2,138</td>
<td>6.1%</td>
</tr>
<tr>
<td>Roberts</td>
<td>1,857</td>
<td>1,978</td>
<td>6.5%</td>
</tr>
<tr>
<td>CRC</td>
<td>1,627</td>
<td>1,705</td>
<td>4.8%</td>
</tr>
</tbody>
</table>

Average Resource Overhead: 4.1%

**REFERENCES**