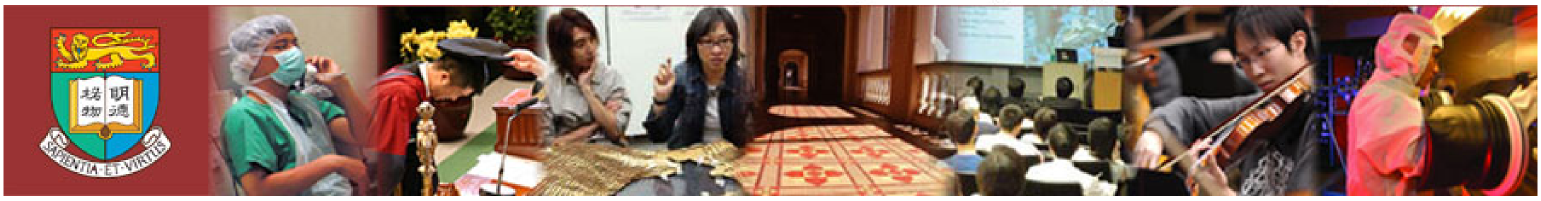


# EXTENDING BORPH FOR SHARED MEMORY RECONFIGURABLE COMPUTERS

*Xun Changqing, Wen Mei, Wu Nan, Zhang Chunyuan* *

Computer School
National University of Defense Technology
Changsha, China
email: {xunchangqing,meiwen,
nanwu,cyzhang}@nudt.edu.cn

*Hayden Kwok-Hay So*

Department of Electrical Engineering
and Electronic
University of Hong Kong
Hong Kong
email: hso@eee.hku.hk

## ABSTRACT

We extend BORPH for shared memory reconfigurable computers in this paper. BORPH is an operating system designed for FPGA based reconfigurable computers. BORPH introduced the concept of hardware process in contrast to software process. With our extension, hardware processes are supported to communicate with other processes based on shared memory. In our system, the program of hardware process is not just hardware design, but the software program running on embedded processor in FPGA. Our experiment shows the overhead of shared memory segments management is acceptable. And with independent virtual memory access, bandwidth of repeated shared memory access is high.

## 1. INTRODUCTION

BORPH[1] is an operating system designed for FPGA based reconfigurable computers, implemented as an extension of the Linux kernel. BORPH introduced the concept of hardware process. As software processes are executed on the system CPU, hardware processes are executed on the reconfigurable hardware. We call the program of hardware process hardware program in this paper. Communications between a hardware process and the rest of the system are accomplished through conventional UNIX inter process communication (IPC) mechanisms.

There is no shared memory between hardware and other processes in the system by default under BORPH. But in FPGA platforms we commonly used, FPGA can access system memory over the PCI Express (PCIE) bus. This means that the system memory can be shared between CPU and FPGA.

*National Nature Science Foundation of China under NSFC No. 61033008, 60903041 and 61103080, Research Fund for the Doctoral Program of Higher Education of China under SRFDP No. 20104307110002, Hunan Provincial Innovation Foundation For Postgraduate under No.CX2010B028, Fund of Innovation in Graduate School of NUDT under No.B100603.

**Table 1**. APIs of software library on NIOS2.

| API | Description |
|---|---|
| AOS_shmget | Get the IPC identifier of a shared memory region. Optionally creating it if it does not already exist. |
| AOS_shmat | Attach an IPC shared memory region to a process. |
| AOS_shmdt | Detach an IPC shared memory region specified by its IPC identifier. |
| AOS_shmctl | Control an IPC shared memory region. |
| AOS_getvaddr | Translate a virtual address to physical address. Deal with page fault if it occurs. |

In this paper, we introduce an extension of BORPH to support shared memory FPGA based reconfigurable computers. A software interface is provided for hardware processes to manage and access shared memory segments. We focus on virtual memory in this work, because it is the key for hardware process to share memory with software processes.

With this extension, hardware process has access to its own virtual memory space. So hardware process can deal with its own input and output independently. This can significantly reduce the overhead of operating system (OS) kernel.

We specify in more detail our goals in Section 2. The implementation detail of our work is presented in Section 3. The experimental setup used to demonstrate our system and the corresponding results are presented in Section 4.

## 2. GOALS

There are two goals of our extension: a) a software interface for hardware processes to manage and access shared memory segments, b) supporting virtual memory access from hardware processes.

### 2.1. A software interface for hardware processes

The first goal of this paper is to provide users with a novel method to design FPGA application using C language. In

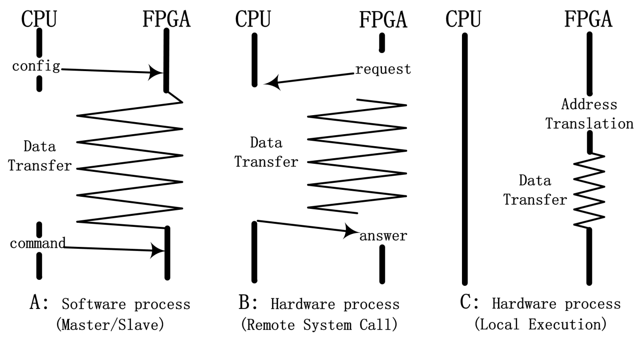

CPU
FPGA
config
Data Transfer
command
A: Software process (Master/Slave)
CPU
FPGA
request
Data Transfer
answer
B: Hardware process (Remote System Call)
CPU
FPGA
Address Translation
Data Transfer
C: Hardware process (Local Execution)

**Fig. 1**. Three types of virtual memory access from hardware process.

our system, the hardware program is not just hardware design, but the software program running on embedded processor in FPGA.

Software program is responsible for communication with the CPU, through a set of APIs we present in this paper, as shown in Table 1. The semantics of the first four APIs are the same to the four of the System V IPC: *shmget*, *shmat*, *shmdt* and *shmctl*. They are used for managing shared memory segments.

Hardware processes using our APIs can communicate with any software processes using System V IPC APIs based on shared memory. It also works for between hardware processes. Return value of *shmat* is a virtual address pointing to the shared memory segment. To support access to shared memory segments from hardware processes, we have to achieve the second goal.

### 2.2. Virtual Memory Access From Hardware Processes

Miljan Vuletic[2] designed window memory management unit (WMU) like memory management unit to support virtual memory access from FPGA. But it is implemented as a hardware module.

We attempt to avoid overhead and complexity by implementing our virtual memory management unit as a software module running on the embedded processor in FPGA. To further discuss the method of virtual memory access in this study, we demonstrate three types of methods as shown in Fig. 1.

Fig. 1A shows a software process using FPGA as accelerators. Through this method, the CPU is responsible for the entire course of virtual memory access. Firstly, CPU configures the FPGA to ensure that the FPGA is ready. Then, CPU starts data transfer for data transfer between system memory and the FPGA. When the transfer is complete, CPU will issue a command to FPGA.

Fig. 1B uses the concept of hardware process, but by a simple method. In concept, virtual memory access is controlled by the FPGA alone. An simple implementation is that FPGA initiative to start the data transfer, but only makes a request to CPU, and CPU is responsible for response and complete the data transfer, and then send a signal to the FPGA.

Fig. 1C is the proposed method. our target is to localize the entire process in the FPGA without interrupting CPU. FPGA do address translation and control data transfer by itself. The Fifth API *AOS_getvaddr* in Table 1 can translate virtual addresses to local addresses. Then data transfer could be easily controlled by a component in FPGA.

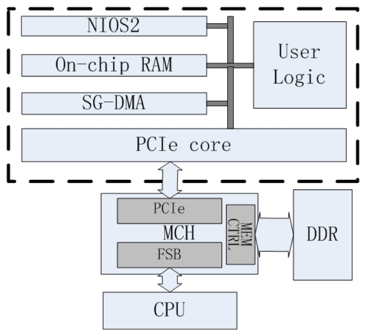


**Fig. 2**. Block diagram of our experimental system.

## 3. IMPLEMENTATION

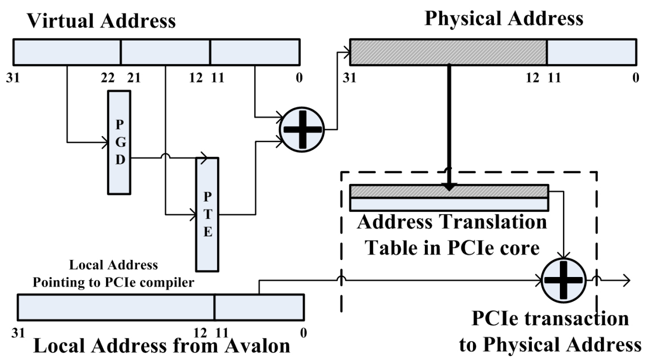
The experimental system consists of a FPGA board based on the Altera Arria II GX device and a PC. The FPGA board is plugged into the PC over PCIE.

Our extension of BORPH is implemented as a Linux kernel module, basic hardware system on FPGA and a software library on NIOS2 (Altera embedded processor). The kernel module is in charged of the management and response to the request of hardware processes.

### 3.1. FPGA On-chip architecture

Fig. 2 shows the block diagram of our hardware platform. The dashed area is implemented on FPGA, and both FPGA and CPU access DDR by memory controller hub (MCH). System on FPGA consists of two parts: basic hardware system and user defined logic. The former part is used to support the running of our system, including NIOS2[3], PCIE core[3], On-Chip RAM, scatter-gather direct memory access (SG-DMA). The latter part is designed according to the specific requirement from users, and all modules would be connected by Avalon bus.

NIOS2 is a popular soft-core provided by Altera. PCIE core is generated by Altera's IP compiler for PCI Express. We configured the compiler as a hardware implementation PCIE 1x endpoint with Avalon-MM interface. Maximum length of burst read is 256 bytes, and burst write is 128 bytes.



Virtual Address
31 22 21 12 11 0
PGD
PTE
Physical Address
31 12 11 0
Address Translation Table in PCIe core
Local Address Pointing to PCIe compiler
31 12 11 0
Local Address from Avalon
PCIe transaction to Physical Address

**Fig. 3**. Local address to host physical address translation.

On-Chip RAM is mainly used as software management on-chip memory of FPGA to enhance the efficiency of hardware processes. SG-DMA is Altera's DMA controller.

### 3.2. Software library on NIOS2

We design a software library on NIOS2 to support the APIs listed in Table 1. The first four APIs execute on CPU remotely based on the message passing mechanism between NIOS2 and CPU. Message passing is implemented through the use of mailbox in PCIE core. We call them remote system call. The last one executes on NIOS2 locally, and it is the key for hardware processes to independently access the virtual memory system. We will discuss the implementation of it in detail.

The available system memory is partitioned into pages typically 4 KB in length. A set of page tables is maintained by OS to specify how virtual addresses are translated to physical addresses. The virtual addresses are divided to three sections, the first two are pointers to page global directory (PGD), page table entry (PTE) respectively, and the last one is offset. PGD and PTE are both stored in kernel space, so NIOS2 can read them through their physical addresses which are equal to virtual addresses minus *0xC0000000*. Note that NIOS2 access the system memory directly for PGD and PTE. NIOS2 didn't copy them to the On-Chip RAM of FPGA.

The *AOS_getvaddr* can tranlate virtual addresses to local addresses by looking up PGD and PTE. Fig. 3 depicts the address translation process. First we look up PGD and PTE according to the virtual address to get the physical address. The high 20 bits of the physical address is the start address of the physical page. Then the start address will be wrote into the address translation table in PCIE core. After this, any access from Avalon to PCIE core will be transformed to PCIE transactions to the physical address. *AOS_getvaddr* actually returns the local address of PCIE core.

Null entry in PGD or PTE means page fault. We deal with this problem like collaborative exception handling (CEH) in EXOCHI[4]. A message about this page fault will be send to CPU automatically when a page fault occurs. CPU will handle the page fault on behalf of NIOS2.

The NIOS2 handles address translation independently instead of CPU. Although NIOS2 runs slower than CPU, the address translation is simple, which makes the longer execution time acceptable compare with overhead of cooperation between NIOS2 and CPU.

### 3.3. memory and page table consistency

Memory consistency must be considered in a heterogeneous system consisting of FPGA and CPU. Memory consistency is not a problem in our system due to the feature of FSB supporting snooping[5].

NIOS2 accesses the page tables in system memory. There is no copy of page tables in our system. So page table consistency is not an issue in our system with memory consistency.

## 4. RESULTS

This paper implemented the extension of the BORPH to achieve two objectives. The experimental system is built on a PC platform: Intel E5200, 4G DDR2 memory. FPGA motherboard applies Arria II GX chip, clock frequency of the on-chip system is 50MHZ. Resource utilization of each component are shown in Table 2. Total FPGA resource usage is relatively low.

This paper carries out performance testing for two functions provided: shared memory managements management (through four remote system call) and virtual memory access. All time units are the FPGA clock cycle.

### 4.1. Remote system call

We use *AOS_shmget* as an example to evaluate the overhead of calling remote system call. Three other calls are similar. Our experiment shows that it costs 1980 clock cycles.

The flow of remote system call is simple. NIOS2 sends a request to the CPU through the Mailbox in PCIE core, the request is issued to the CPU with the response time cost about 600 clock cycles. Followed this, the actual system call running on CPU costs about 800 clock cycles. Lastly, CPU issued a request via PCIE to NIOS2 and return the results when complete.

The remote system call is relatively inefficient. However, this does not affect the usefulness of inter-process communication through shared memory we provide, because once the process attaches the shared memory segment to its virtual address space, remote system call is no longer needed to access shared memory.

**Table 2**. Resources utilization of each component.

| Resource | Avail. | NIOS2 | PCIE core. | On-Chip MEM | SG-DMA |
|---|---|---|---|---|---|
| ALUT | 99280 | 995(1%) | 4491(4%) | 0(0%) | 787(0.7%) |
| Block MEM (bits) | 6727680 | 46464(0.6%) | 111328(1.6%) | 32768(0.4%) | 278756(4%) |

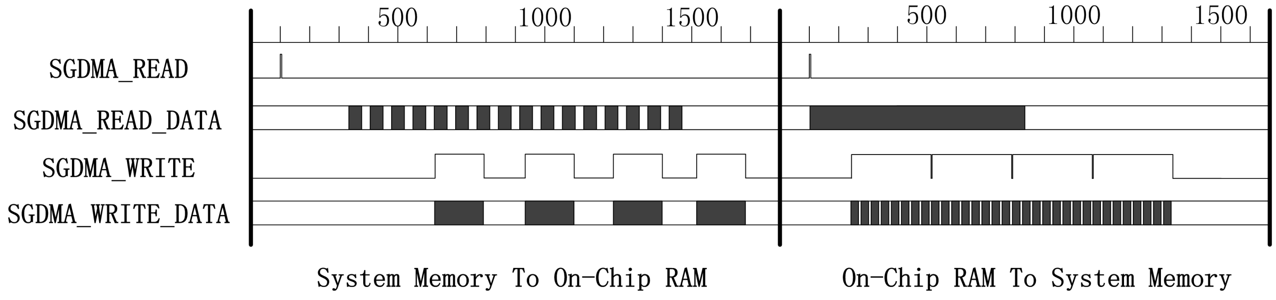


**Fig. 4**. Signals of SG-DMA controlling Data Transfer.

## 4.2. Virtual memory access

We evaluate the overhead of single physical page(4KB) access. Access to single physical page consists of two steps: *AOS_getvaddr* for address translation (including possible page fault handling) and the actual data transfer. The overhead of former is basically fixed at any time with slight changes.

Address translation consists of two system memory access for PGD and PTE from NIOS2, and costs about 200 cycles. Overhead of page fault handling is similar to that of a remote system call, which is about 1500-2000 cycles. Due to the fact that during each execution of a hardware process, the physical addresses it accesses are fixed, the page fault also occurred only for the first time. So the cost of *AOS_getvaddr* in practical will be relatively small.

We evaluate the efficiency of follow-up data transfer in detail, because it is need to be repeated many times. We define load data from the system memory to the On-Chip RAM as read operation, otherwise known as the write operation.

Data transfer between the system memory and the On-Chip RAM is equal to data transfer between PCIE core and On-Chip RAM. So we first simply use the *memcpy* to copy data between On-Chip RAM and PCIE core. This method cannot make use of burst transfer capacity of the PCIE core. The payload of each PCIE transaction is only 4 bytes, so the efficiency is very low.

Then SG-DMA is used to improve the efficiency. Fig. 4 shows the signals of read and write ports of SG-DMA. From top to bottom are respectively the signals of read, read data, write and write data. Left and right sides of Fig. 4 shows the result of a read and write operation respectively. For a read operation, 16 PCIE read transactions with 256 bytes payload are needed, while for a write operation, 32 PCIE write transactions with 128 bytes payload are needed. The write buffer of SG-DMA is set to be less than the read buffer. So the overlap between read and write is allowed as shown in Fig. 4. It is clear that efficiency is improved greatly.

Bandwidth of system memory access showed in Fig. 4 is the highest we can achieve now. There are two further optimization for better performance. First, increasing the clock frequency of FPGA on-chip system could naturally improve the bandwidth. Second, a higher transfer efficiency can obtained by a optimized PCIE core, because the maximum payload of single PCIE transaction is 4KB which is much bigger than that the current PCIE core can support.

## 5. REFERENCES

[1] R. Hoyden Kwok Hay So, Brodersen, "Improving usability of fpga based reconfigurable computers through operating system support," in *Proc. IEEE Int. Conf. Field Programmable Logic and Applications.*, Apr. 2006, pp. 349–354.

[2] I. P. Vuletic M., Pozzi L., "Virtual memory window for application-specific reconfigurable coprocessors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 8, pp. 910–915, July 2006.

[3] *http : //www.altera.com.cn.*

[4] G. N. C. Perry H. Wang, Jamison D. Collins, "Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system," in *Proc. ACM SIGPLAN Conf. Programming language design and implementation.*, vol. 42, no. 6, June 2007.

[5] L. F. Wissam Hlayhel, Jacques Collet, "Implementing snoop-coherence protocol for future smp architectures," in *Proc. IEEE Int. Euro-Par Conf. Parallel Processing.*, 1999.