pvFPGA: Accessing an FPGA-based Hardware Accelerator in a Paravirtualized Environment

Wei Wang
Computer Architecture Research Group, University of Ottawa, Ottawa, Ontario, Canada
wwang021@uottawa.ca

Miodrag Bolic
Computer Architecture Research Group, University of Ottawa, Ottawa, Ontario, Canada
mbolic@eecs.uottawa.ca

Jonathan Parri
Computer Architecture Research Group, University of Ottawa, Ottawa, Ontario, Canada
jparri@uottawa.ca

ABSTRACT

In this paper we present pvFPGA, the first system design solution for virtualizing an FPGA-based hardware accelerator on the x86 platform. Our design adopts the Xen virtual machine monitor (VMM) to build a paravirtualized environment, and a Xilinx Virtex-6 as an FPGA accelerator. The accelerator communicates with the x86 server via PCI Express (PCIe). In comparison to the recent accelerator virtualization solutions which primarily intercept and redirect API calls to the hosted or privileged domain's user space, pvFPGA virtualizes an FPGA accelerator directly at the lower device driver level. This gives rise to higher efficiency and lower overhead. In pvFPGA, each unprivileged domain allocates a shared data pool for both user-kernel and inter-domain data transfer. In addition, we propose a new component, the coprovisor, which enables multiple domains to simultaneously access an FPGA accelerator. The experimental results have shown that 1) pvFPGA achieves close-to-zero overhead compared to accessing the FPGA accelerator without the VMM layer, 2) the FPGA accelerator is successfully shared by multiple domains, and 3) distributing different maximum data transfer bandwidths to different domains is achieved by regulating the size of the shared data pool at the split driver loading time.

Categories and Subject Descriptors

C.3.3 [**Computer Systems Organization**]: Special-purpose and Application-based Systems

General Terms

Design, Experimentation, Verification

Keywords

FPGA; hardware accelerator; paravirtualization; pvFPGA; coprovisor; shared data pool

1. INTRODUCTION

Cloud computing, which refers to both the applications delivered as services over the Internet and the hardware and systems software in data centers that include these services, has taken center stage in information technology in recent years [1]. As a key component of cloud computing, virtualization technology has received tremendous attention. With virtualization technology, the underlying hardware resources can be shared by multiple virtual machines or domains with each running its own operating system (OS). This gives rise to higher hardware utilization and consequently lower power consumption. The Virtual Machine Monitor – VMM (also referred to as a hypervisor), is responsible for isolating each running instance of an OS from the physical machine. The VMM translates or emulates special instructions of a guest OS. VMMs are classified into two types by Goldberg [12]. Xen [2] is a classic example of widely used type 1 VMMs known as bare-metal or native VMMs. These VMMs run directly on the underlying hardware. Type 2 VMMs, such as VMware Workstation [3] and KVM [18] which reside in an existing OS environment are denoted as hosted VMMs.

Graphical processing unit (GPU) and Field Programmable Gate Array (FPGA) based hardware accelerators have been gaining popularity in the server industry. Accelerators speed up computationally-intensive parts of an application. Successfully and efficiently adding hardware accelerators to virtualized servers will bring cloud clients apparent speedup for a wide range of applications. GPUs are inexpensive, and typically programmed using high level languages and APIs which abstract away hardware details [4]. Despite many challenges of making GPUs a shared resource in a virtualized environment, many papers [7, 11, 14, 19, 22, 24] have succeeded. FPGAs are known to outperform GPUs in many specific applications [4, 6]. Ability to perform partial run-time reconfiguration [9] is an important distinguishing feature of FPGAs. To date, some effort has already been put into FPGA virtualization [8, 13, 15, 20, 23], but the research remains at the multitasking level on a single OS to the best of our knowledge.

In this paper, we move a step forward in the virtualization of an FPGA for its use as a hardware accelerator by the VMM. We have adopted the Xen VMM to build a paravirtualized environment. The FPGA accelerator communicates with the server via PCI Express (PCIe). Our work in this paper mainly focuses on: 1) designing a mechanism that allows a user process from a guest domain to access the FPGA accelerator with minimal overhead, 2) enabling processes from multiple guest domains to simultaneously request access to the shared FPGA accelerator, and 3) enabling special domains (or VIP domains) to finish their requests faster than ordinary domains when they contend for accessing the shared FPGA accelerator.

The remainder of the paper is organized as follows. We begin in Section 2 with an overview of the Xen-based paravirtualization technology. In Section 3, we introduce our pvFPGA design. The implementation and evaluation of the pvFPGA design is revealed in Section 4. Section 5 reviews related work. Ultimately, we conclude this paper and introduce our future efforts in Section 6.

2. AN OVERVIEW OF THE XEN VMM

As the first open-source paravirtualizing VMM, Xen was first released in 2003 mainly for x86 platforms [2]. There are four execution priority levels (termed as rings) on x86 platforms, with ring 0 set as the highest priority. In a paravirtualized environment, an OS needs to be modified to run in ring 1. The Xen VMM exclusively runs in ring 0 guarding accesses to all privileged operations and hardware resources. Meanwhile, the Xen VMM has offered a hypercall mechanism which serves as the only method for these running OSes to interact with the Xen VMM to request privileged operations, i.e. updating page tables. Figure 1 shows an overview of the Xen-based paravirtualized environment.

Xen refers to each running virtual machine as a domain. Xen supports only one privileged domain, domain 0 (Dom0), which is usually the driver domain. Multiple unprivileged domains (DomU) are also supported. Typically, all the unprivileged domains are not given direct access to the underlying hardware; instead they require the use of a split device driver model. As shown in Figure 1, a virtual frontend driver in a DomU communicates with the related virtual backend driver in Dom0 (driver domain), which is usually achieved by using shared memory. The latter then forwards the received I/O requests to the corresponding real device driver. The Xen VMM provides the event channel mechanism by which the frontend and backend drivers notify each other when the I/O requests or responses have been sent. Recently, Xen has added direct I/O access (also known as device passthrough) support for a domain, such as PCI passthrough. The passthrough method is capable of giving a particular domain near-native I/O performance, but it violates the concept of sharing in virtualization. It also causes some security problems on systems without an input/output memory management unit (IOMMU) [33].

Each domain has its own grant table which is shared with the Xen VMM. The table is a data structure storing the information that is used for memory sharing between domains, such as which operation is allowed for the grantee domain to perform on the granter's offered memory. An entry of such a data structure in a grant table is identified by a grant reference which is an integer index. When one domain wants to share its memory with another domain, a grant reference corresponding to the data structure describing that shared memory is required to be passed to the grantee by the granter via some out of band mechanism, such as XenStore.

The default and widely used scheduler in the Xen VMM is a credit scheduler which focuses on allocating CPU resources in a fair manner to each domain according to their pre-assigned weights. The scheduler adopts vCPUs (virtual CPU) as scheduling entities, and each domain can be assigned one or more vCPUs. The default scheduling time quantum is 30ms; that is, scheduling decisions are made every 30ms. The scheduler debits the credits of each running domain on a tick period (10ms) basis. A domain can remain in the UNDER FIFO queue, but will be moved to the tail of the UNDER FIFO queue if it still has credits remaining. Domains that have consumed all of the allocated credits will be put into the OVER FIFO (first-in-first-out) queue at the following scheduling point, which means that they are over scheduled. Once the sum of all the active domains becomes negative, the scheduler will allocate new credits to all the domains at the next scheduling point according to their weight. Domains in the OVER queue will not be selected to run unless no domains in the UNDER queue are ready to run [21]. Therefore, some domains will use more than their fair share of processor resources only on the condition that the processor would otherwise have been idle.

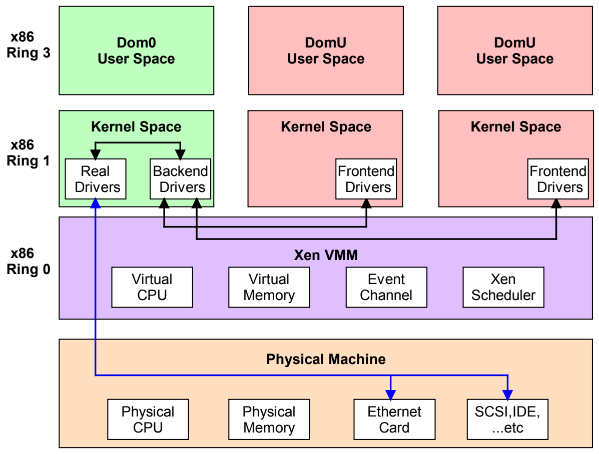


Figure 1. An overview of Xen-based paravirtualized environment

3. SYSTEM DESIGN

The entire system design can be split into two major parts: FPGA accelerator design and FPGA virtualization on an x86 server. The accelerator work is a conventional FPGA hardware design, which is normally implemented using HDL (Hardware Description Language) and/or IP cores. The latter work is software-related development that involves the design of a frontend and backend driver along with a real device driver for the FPGA accelerator. This includes the proposed coprovisor design. The overall layout of pvFPGA system design is shown in Figure 2. Each part of our work will be detailed in the following two subsections, 3.1 and 3.2.

3.1 FPGA Hardware Accelerator Design

We adopted the PCIe interface [30] as the communication channel, and used the direct memory access (DMA) technique for efficient transfer of data to and from the host server memory. Unlike traditional ISA (Industrial Standard Architecture) devices, there is no central DMA controller for PCIe components. To address this, we designed a DMA controller on the FPGA accelerator. This is called the bus mastering DMA (also known as first-party DMA), which describes the ability of the DMA controller residing in the PCIe device to initiate PCIe transactions (memory read and write transactions). With only one DMA channel [26], we need to utilize additional memory (e.g. DDRII memory) to store data when the computing procedure involves large amounts of data. We used the DDRII memory controller IP core [31] from the Xilinx Corporation. The overall design of the FPGA accelerator is shown in Figure 3(a). In the figure, same

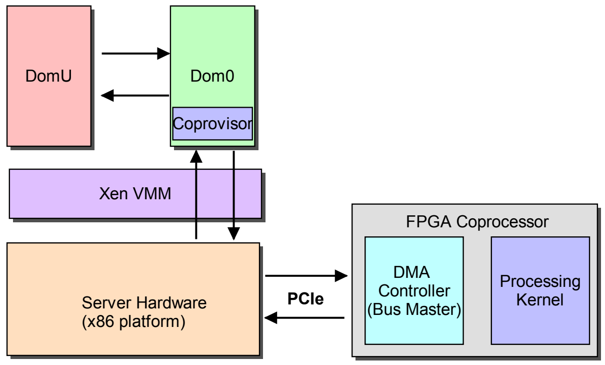


Figure 2. The layout of pvFPGA design

color modules operate at the same frequency.

We have further upgraded our design by multiple DMA channels. The off-chip DDRII memory (bottom part of Figure 3(a)) can be removed for majority of the applications, because we can do DMA reads from the server memory and DMA writes to the server memory simultaneously. In other words, we can take advantage of a streaming pipeline.

Figure 3(b) illustrates the simplified design using two DMA channels [32], and Figure 3(c) shows the pipelined operations. The overall procedure is conducted in three operations: DMA read, computation and DMA write. The latency for completing each of the three operations is identical in Figure 3(c), but in practice, it is usually not the same.

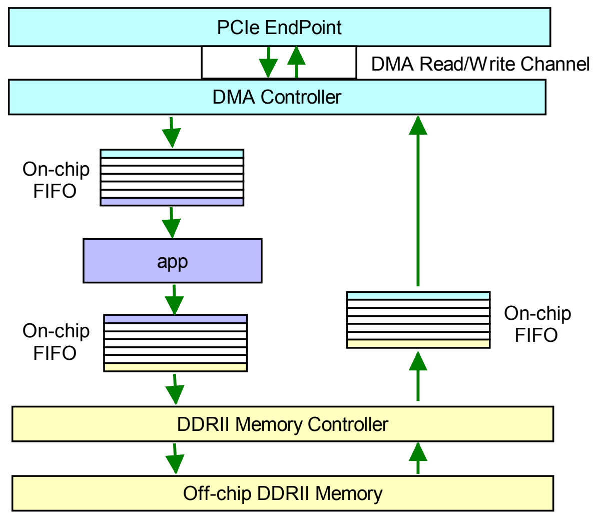
3.2 FPGA Virtualization

3.2.1 *Data transfer*

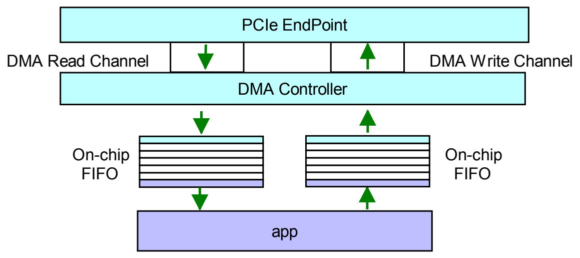
Using an FPGA hardware accelerator requires transfers of tens of Kilobytes to Gigabytes of data between the server and the FPGA accelerator. The overhead, compared with a native machine without the VMM layer, is caused by inter-domain communication latency. This latency is the time consumed by transferring large amounts of data from a DomU to Dom0 via the split device driver model supported by the Xen VMM. The aim of this part of the design is to access the FPGA accelerator from a DomU with low time overhead. In addition to inter-domain communication latency, we also have to consider how the data could be efficiently transferred from a user application (residing in user space) to the frontend driver (residing in kernel space) on a DomU.

The challenge of designing efficient inter-domain communication is not unique to FPGA virtualization. Some shared memory based techniques, such as IVC [16], Xway [17], XenLoop [25], and Xensocket [28], have tackled the problem for network packets communication, but their designs are specific to network packets. The shared memory mechanism has also been used in recent GPU virtualization solutions [14, 24] for inter-domain communication resulting in low overhead. These solutions are specifically designed for intercepting and redirecting API calls for a GPU accelerator. Here, we introduce a shared-memory mechanism for pvFPGA that focuses on low overhead data transfer for an FPGA accelerator. The comparison of pvFPGA with recent GPU virtualization solutions is detailed in Section 5.

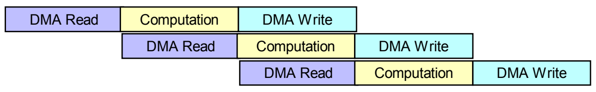
Figure 4 shows the inter-domain communication design of pvFPGA. A DomU kernel allocates a group of 4KB memory pages (we refer to this as a "data pool") which are reserved for



a) Design with one DMA channel



b) Design with two DMA channels



c) Pipelined operations

Figure 3. Design of an FPGA accelerator

data transfer. Since the DMA controller referenced in our FPGA accelerator design supports a scatter/gather function, these 4KB memory pages allocated in the DomU kernel space are not necessarily physically continuous. A DomU's user process can map the data pool into its virtual address space so that it can directly operate on the shared memory pages. In order to enable a process to map the data pool, the allocated pages are represented as one device file in the */dev* directory. We write the following system call implementations for user processes to operate on the data pool:

1. *mmap()*

In this function, we remap all the allocated 4KB physical pages to the calling process's virtual address space according to the order they are allocated. The start address of the mapped virtual memory area will be returned. After this call, the user process can directly access the data pool.

2. *write()*

This function implements two actions: sending a request to Dom0, and putting the calling process into the Sleep state. This is an interface that a user process uses to notify the DomU kernel that the data is ready. When the calling process is put into

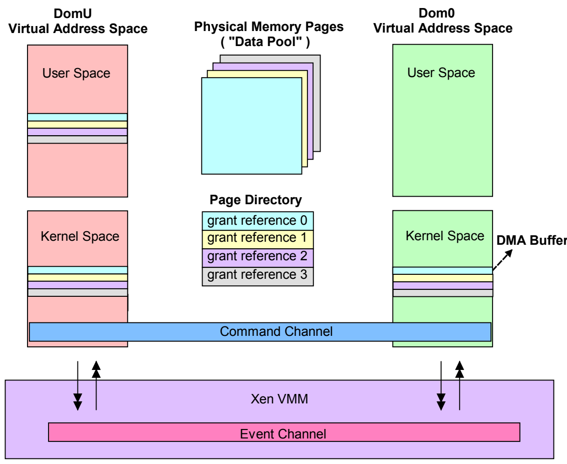


Figure 4. Inter-domain communication design of pvFPGA

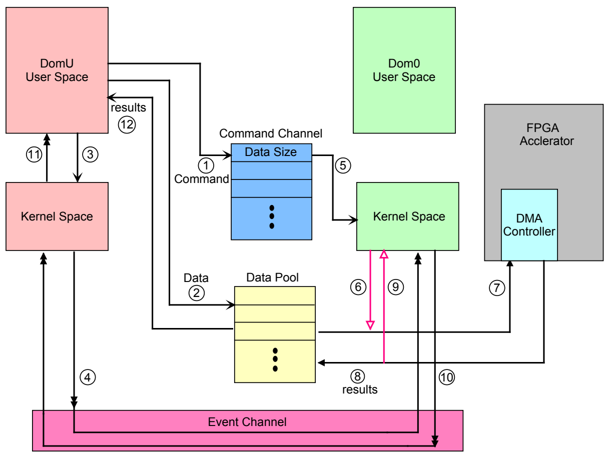


Figure 5. Dataflow in pvFPGA

Sleep state, and if there are other processes that are ready to run in the domain, a process context switch will be implemented.

Meanwhile, the data pool is also shared with the Dom0 kernel through the grant table mechanism. To achieve the sharing of a data pool between domains, the grant references of these shared memory pages in a data pool are filled in an extra shared memory page which acts as a page directory. The directory is shared with Dom0 first. Once Dom0 gets the directory page, it is ready to map the offered data pool. Lastly, the shared memory pages in the data pool, acting as a DMA buffer, are exposed to the bus master DMA controller residing in the FPGA accelerator.

Similar to the data pool, the command channel is a memory page shared between a DomU (also mapped by a user space process with the same method) and Dom0. It is used for transferring command information. Our design currently uses the command channel for transferring only one piece of command information: the size of data in the data pool that needs to be transferred to the FPGA accelerator. For example, a DomU has a 4MB data pool, but it may request only 256KB data to be transferred for processing on the FPGA accelerator.

Figure 5 presents the dataflow in pvFPGA. The steps are described as follows:

1. an application specifies the data size in the command channel that is mapped to its address space through the “mmap()” call;
2. the application directly puts data in the shared data pool that is mapped to its address space;
3. the user application notifies the frontend driver in the DomU kernel space that data is ready and then goes to the Sleep state (this is achieved by calling “write()”);
4. the frontend driver in the DomU kernel space sends an event to the backend driver in the Dom0 kernel space;
5. the frontend driver passes the request to the device driver in the Dom0 kernel space, and the device driver sets the DMA transfer data size according to the parameter obtained from the command channel;
6. the device driver initiates the start of the DMA transfer in the FPGA accelerator;
7. the DMA controller transfers all the data to the FPGA accelerator in a pipelined way to do computations;
8. the DMA controller transfers the results of computations back to the data pool;
9. the DMA controller sends an interrupt to the device driver when all the results have been transferred to the data pool;
10. the backend driver sends an event to notify the frontend driver that the results are ready;
11. the frontend driver wakes up the user application;
12. the user application fetches the results from the data pool.

All of the memory-sharing procedures are finished during the frontend/backend driver loading time. Therefore, the only overhead caused by the Xen VMM is the two event notifications for the inter-domain data transfer, one for notifying Dom0 that data is ready to be sent to the FPGA accelerator and another for notifying DomU that the results are ready to be fetched.

3.2.2 Coprovisor

The Xen VMM scheduler is only responsible for controlling CPU access, while the backend drivers provide some means of regulating the number of I/O requests that a given domain can perform [5]. We propose a component that we call coprovisor for pvFPGA which multiplexes requests from different domains accessing the FPGA coprocessor. For GPU virtualization, the multiplexing work is realized in user space due to the lack of standard interface at the hardware level and driver layer. More precisely, the multiplexer and scheduler are put on the top of the CUDA runtime or driver APIs [14, 24]. In our case, the coprovisor is able to perform multiplexing directly at the accelerator driver layer in Dom0.

The architecture of the coprovisor is shown in Figure 6. It consists of four parts: Request Inserter, Scheduler, Request Queue and Request Remover. A DomU notifies Dom0 accessing the FPGA accelerator via an event channel. The Request Inserter, which is responsible for inserting requests from DomUs into the Request Queue, is invoked when an event notification is received at the backend driver. When a request has been serviced, an interrupt from the FPGA accelerator notifies the Request Remover to remove the serviced request from the Request Queue. The Scheduler is responsible for scheduling access requests for the FPGA accelerator through the accelerator device driver. The present scheduling algorithm used by the scheduler is FCFS (first-come, first-served); that is, requests from DomUs are extracted in an orderly manner by the scheduler. Either when a new request is inserted in the Request Queue or a serviced request is removed from the queue, the Scheduler will be invoked to check if the FPGA accelerator is idle and if there is a pending request in the Request Queue. If so, the head request of the Request Queue will be scheduled to the FPGA accelerator. A request is adopted as a scheduling entity by the coprovisor; that is, requests from DomUs

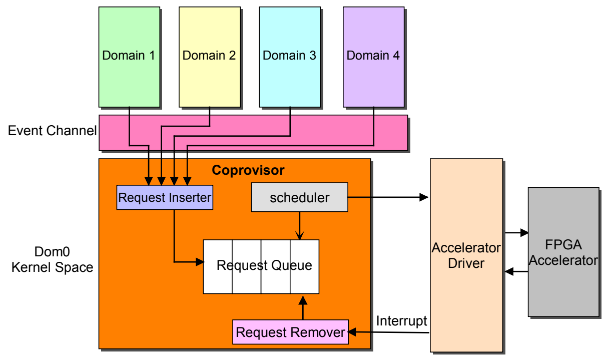


Figure 6. The coprovisor design of pvFPGA

are scheduled in FCFS to access the FPGA accelerator.

The size of shared data pool implies the maximum data transfer bandwidth. For example, a DomU (D1) assigned a 4MB data pool can transfer a maximum of 4MB of data for each request to the FPGA accelerator. A DomU (D2) assigned a 512KB data pool can transfer a maximum of 512KB data per request. When the two DomUs contend for a shared FPGA accelerator and they need to send more than 512KB of data, D2 is slower because it needs to send more requests to complete the computations. To provide DomUs with different maximum data transfer bandwidths, we can regulate the size of the shared data pool in each DomU's frontend driver and the Dom0's backend driver at the frontend/backend driver loading time.

4. IMPLEMENTATION & EVALUATION

4.1 Experimental Platform

Our experiments were conducted on a server equipped with an Intel® Xeon Processor W3670 running at 3.2GHZ and 4 GB of DDR3 main memory. A Xilinx Virtex-6 based evaluation board is used as a hardware accelerator, which communicates with the server via 4-lane PCIe (Generation 2). Xen-4.1.2 is used as the VMM. Ubuntu 12.04LTS is used as Dom0 and Ubuntu 10.04LTS as the DomUs. We built four DomUs named Lucid1, Lucid2, Lucid3, and Lucid4.

In the following experiments, we first estimate the virtualization cost with a simple loopback application, then we verify the validity of pvFPGA with a Fast Fourier Transform (FFT) benchmark. Lastly, the functionality of the coprovisor is evaluated by the four DomUs' simultaneous requests to access the shared FPGA accelerator.

4.2 Overhead Evaluation

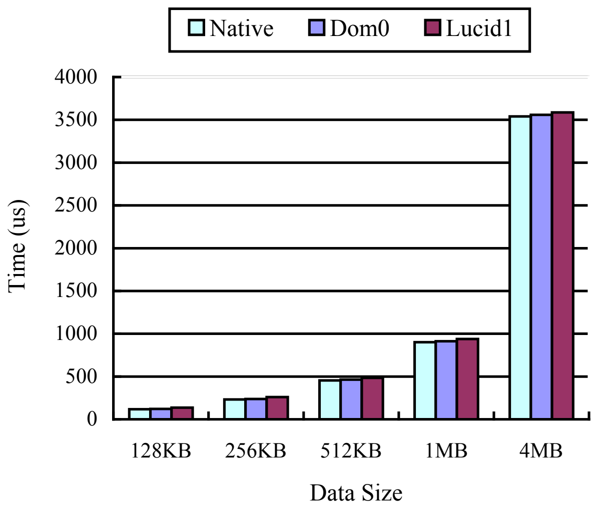
In order to estimate the overhead caused by the virtualization layer, we use a loopback application in the FPGA accelerator. This application simply returns unmodified data it receives. The DMA read and write operations in Figure 3(c) are implemented with a 4KB (Page Size of Linux OS) data block size. The evaluation is implemented in:

1. a native Linux environment without the Xen VMM,

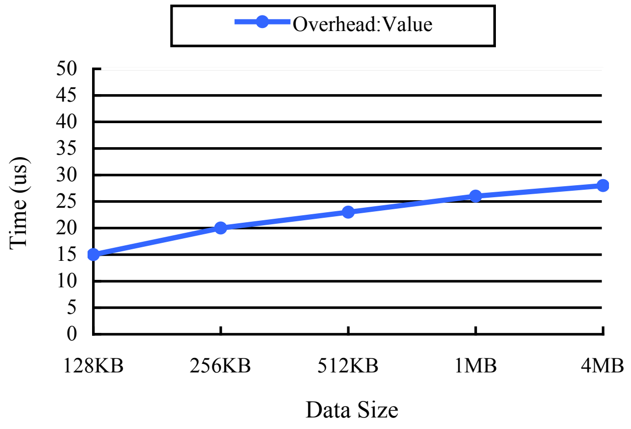
2. a virtualized Linux environment but with direct access to the FPGA accelerator (Dom0),

3. a guest domain in pvFPGA (Lucid1).

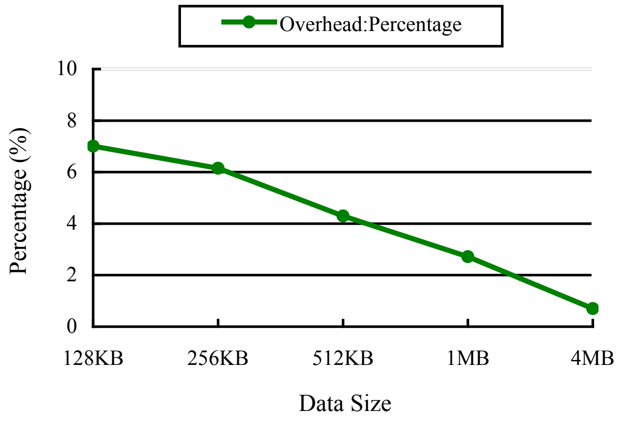
The data pool size is increased from 128KB to 4MB in this experiment. The results are averaged across 20 experiments. Figure 7(a) shows that there is no apparent performance



a) Tunaround time



b) Virtualization overhead



c) Virtualization overhead compared to turnaround time

Figure 7. Overhead evaluation with a loopback application

difference between native Linux and Dom0 (driver domain) in terms of accessing the accelerator, which is also revealed in paper [14]. We choose Dom0 as our base case for the following evaluation. We summarize the overhead caused by the virtualization layer by comparing the turnaround time of servicing a request sent from Dom0 and from Lucid1 separately. The overhead time shown in Figure 7(b) varies between 15us to 30us while the data pool size grows from 128KB to 4MB. The overhead time remains stable with an increasing data pool size, since the only overhead caused by the Xen VMM is the two event notifications for the inter-domain data transfer. Correspondingly, as shown in Figure 7(c), compared with the turnaround time of sending 4MB data, the overhead time is close to zero percent.

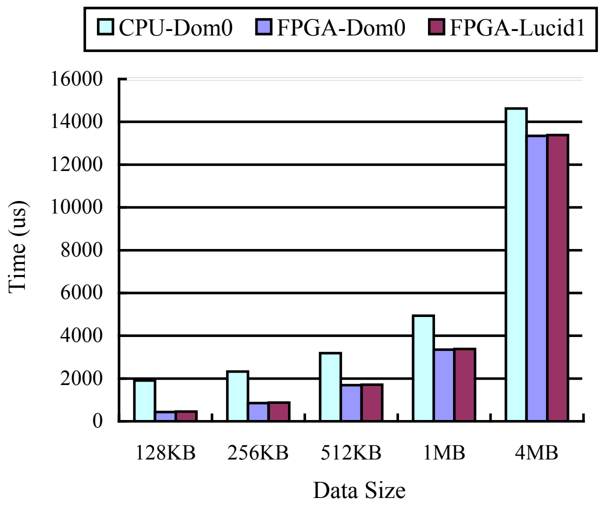
4.3 Verification with Fast Fourier Transform

The second test involves pvFPGA verification with a real world accelerator example. FFT is an algorithm used in a wide range of applications such as signal processing, medical imaging, petrochemical exploration, and defense applications [27]. A benchmark based on Xilinx FFT IP core [29] for accelerating 256-Point floating point (single precision) FFT is used in this experiment. The FFT benchmark module operates at 250MHz. Similarly, data is transferred by DMA in a pipelined way with 4KB per block. Executing the FFT benchmark requires at least 2KB data (1KB real number and 1KB complex number) to be transferred to the FPGA accelerator, so one block of data requires two FFT computations. In this experiment, we send 128KB to 4MB of floating point data to compute FFT on the accelerator.

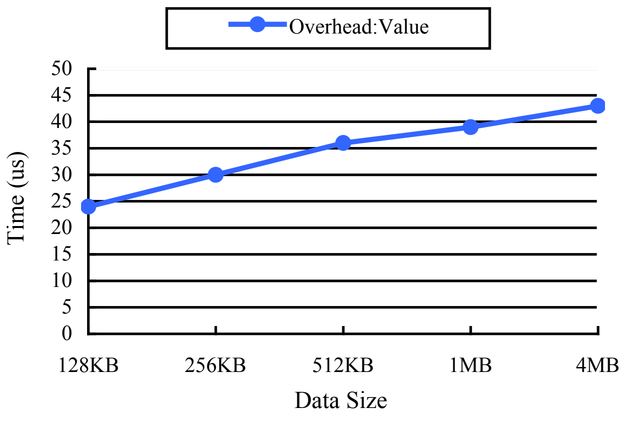
We have verified that all the results received from the FPGA accelerator in a DomU's application are accurate. Figure 8(a) shows the turnaround time that is required for computing FFT with different data sizes on a CPU and the FPGA accelerator including the communication overhead. We send requests to the FPGA accelerator for FFT computations from both Dom0 and a DomU (Lucid1) to verify the overhead. The virtualization overhead is shown in Figure 8(b). In theory, the overhead time should be a fixed value, since the only overhead caused by the Xen VMM is the two event notifications for the inter-domain data transfer. When more data is transferred to the FPGA, more fluctuations occur in the measurement. In Figure 8(b), the measured overhead time is around 10us larger than that in Figure 7(b). This is because using an FFT benchmark has larger turnaround time than using a loopback benchmark, and the time measured with the FFT benchmark has more fluctuations. However, this increase is very small especially for large data pool sizes. As shown in Figure 8(c), the overhead is close to zero percent when a data pool is allocated with 4MB size or larger.

The baseline code we choose to run on a CPU is FFTW3. FFTW3 is a free collection of fast C routines for computing FFT [10], and it is prevalently used among researchers. Using FFTW3 requires an initialization at the beginning, which creates a plan for an array that will be used for FFT computations. Then this array can be repeatedly used if the user needs to perform multiple FFT computations. The influence of the FFTW3 initialization overhead is minimized as the data size grows.

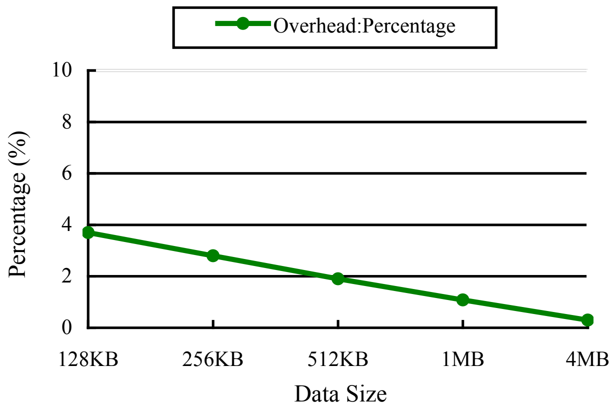
The turnaround time of completing an FFT computation on an FPGA accelerator consists of computation latency and communication latency. The time of completing an FFT computation on CPUs only consists of computation latency. However, we can still benefit from using an FPGA accelerator for computations in most cases, because the computation latency on an FPGA accelerator is usually much smaller than that on CPUs due to its parallel and pipelined processing. Thus, the overall



a) Turnaround time

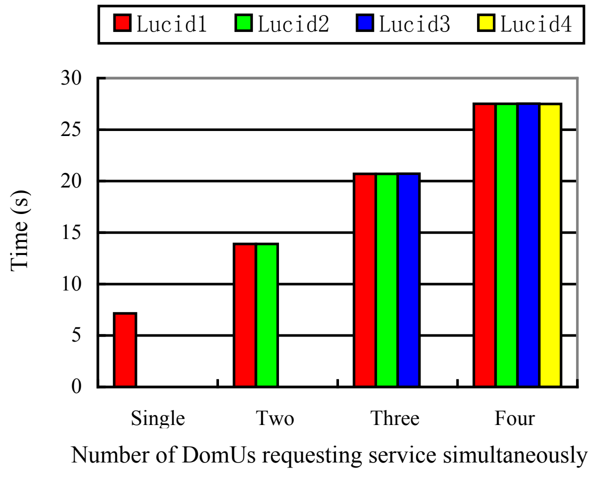


b) Virtualization overhead

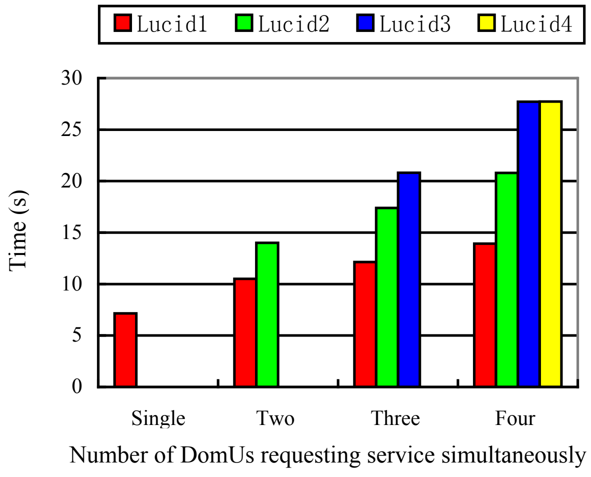


c) Virtualization overhead compared to turnaround time

Figure 8. Verification with an FFT application



a) Identical data pool size in DomUs



b) Different data pool sizes in DomUs

Figure 9. Evaluation of the coprovisor

latency (computation plus communication latency) of using an FPGA accelerator can be shorter than that of using CPUs for computations. As shown in Figure 8(a), the speedup of FFT computations on the FPGA accelerator is over 2 fold when 128KB data is required for FFT computations compared with that on CPUs. The speedup remains around 1.1x (14688us/13348us) with 4MB data. This is because the FFTW3 initialization overhead is minimized when larger data is required for FFT computations. To gain a larger speedup, further optimizations of the FFT computation core on the FPGA accelerator are required. Faster communication channels, such as a PCIe interface with more lanes provide another improvement alternative. This will minimize the communication overhead between the FPGA accelerator and the server.

4.4 Coprovisor Evaluation

Lastly, we evaluate the functionality of the coprovisor, which multiplexes multiple DomUs to access the FPGA accelerator. Each of the four DomUs (Lucid1, Lucid2, Lucid3 and Lucid4) allocates a 1MB size data pool, which is shared with Dom0 for inter-domain data transfer. We still use the FFT benchmark here for the evaluation. The evaluation task assigned to the DomUs requires them to finish 2GB of data FFT computations on the FPGA accelerator. We let all the applications in the four DomUs set their requested data size in each request equal to the size of their allocated data pool size (using maximum data transfer bandwidth) through the command channel. Thus, each DomU needs to send 2048 requests to complete the FFT computations.

The turnaround time of four DomUs' accessing the FPGA accelerator simultaneously for FFT computations with 2GB of data is shown in Figure 9(a). When a DomU accesses the FPGA accelerator individually, the turnaround time for completing FFT computations with 2GB of data is approximately 7 seconds, whereas all of the four DomUs equally spend approximately 27.5 seconds when they contend for one FPGA accelerator.

In order to distribute different maximum data transfer bandwidths to DomUs in pvFPGA, we assign different data pool sizes to each DomU. In this experiment, we assign Lucid1, Lucid2, Lucid3 and Lucid4 with 1MB, 512KB, 256KB and 256KB data pool sizes respectively, separately in each kernel space. Also, each DomU uses their maximum data transfer bandwidth in this experiment. Figure 9(b) shows the turnaround time of completing the evaluation task by assigning different data pool sizes to the DomUs. When all the four DomUs request access to one FPGA accelerator simultaneously, Lucid1 gets serviced two times faster than Lucid3, while Lucid3 and Lucid4 spend equal time (approximately 27.5 seconds) getting their requests serviced because of their equally assigned data pool sizes. As shown, the overall turnaround time of Figure 9(b) equals that of Figure 9(a), because their overall data sizes for FFT computations are the same.

5. RELATED WORK

A direct comparison between pvFPGA and the recent GPU virtualization solutions may seem unfair, since GPUs are designed for general purpose computing. But pvFPGA is inspired by some of the techniques used in GPU virtualization solutions in some respects. In this section, we compare the difference only in terms of the virtualization techniques being used.

Due to the limited knowledge of GPU hardware specifications, it is not feasible to realize GPU virtualization at the low device driver level which leads to higher efficiency and lower overhead. The current solutions used for GPU virtualization, such as GViM [14], vCUDA [24], gVirtuS [11], primarily intercept user space API calls from frontend domains and redirect them to the backend domain. The redirection of virtual CUDA API calls to real API calls is eventually accomplished in the Dom0 user space. GViM adopts XenStore (shared memory) for inter-domain data transfer, which generates relatively low overhead. vCUDA also introduces the shared memory mechanism for their newest version, VMRPC. VMRPC achieves lower overhead (reduced from around 11% to 4%) with its Lazy RPC (Remote Procedure Call) which aims at batching specific RPCs thereby reducing the number of expensive world switches (context switches between different domains). gVirtuS is a VMM independent solution for a cluster environment, and the efficiency of its inter-domain communication relies on the choice of VMM. However, efficiently multiplexing GPU devices for multiple domains or virtual machines in a virtualized environment is cumbersome. vCUDA uses working/service threads (introduced in [24]) to enable requests to be concurrently executed on the GPU accelerator, regardless of whether they come from the same domain or different domains. A resource sharing framework as an extension of gVirtuS has been proposed in [22]. Authors of [22] create a virtual process context to consolidate different applications (may come from different domains) into a single application context to time share or space share streaming multiprocessors (SMs) in a GPU accelerator. On the other hand, GViM still relies on the Xen credit scheduler to manage a GPUs resource allocation.

Both the above GPU virtualization solutions and pvFPGA use shared memory for inter-domain data transfer. In contrast to the GPU virtualization solutions, pvFPGA virtualizes the FPGA accelerator directly at the FPGA device driver layer. Compared with their low overhead solutions, the overhead in pvFPGA is close to zero with a data pool of 4MB or larger size. Moreover, none of these GPU virtualization solutions devise a GPU multiplexing scheme supplying DomUs with different maximum data transfer bandwidths. The coprovisor easily achieves the multiplexing of the FPGA accelerator at the device driver layer, and pvFPGA is capable of offering different maximum data transfer bandwidths for DomUs through regulating the size of each shared data pool in the frontend/backend driver.

6. CONCLUSION AND FUTURE WORK

In this paper, we have proposed pvFPGA which is the first system design solution for virtualizing FPGA-based hardware accelerators. In pvFPGA, each DomU allocates a shared data pool for both user-kernel and inter-domain data transfer. Our experiments have demonstrated that pvFPGA offers close to zero time overhead caused by the virtualization layer. We have also proposed a new component, the coprovisor, which enables multiple domains to access the FPGA accelerator simultaneously. DomUs can be assigned different maximum data transfer bandwidths through regulating the size of their data pools.

As a part of future work, we plan to introduce partial reconfiguration technology to pvFPGA. With partial reconfigurability, accelerator applications can be dynamically downloaded to the FPGA accelerator based on user application requirements. In addition, our future research direction also includes making pvFPGA suited for generous purpose computing so that various applications can be accelerated on an FPGA accelerator in a virtualized environment.

7. REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. University of California, Berkeley, Tech. Rep. UCB-EECS, February 2009.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In Proc. of the 19th ACM Symposium on Operating Systems Principles, pp. 164-177, 2003.

[3] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, E. Y. Wang. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. ACM Transactions on Computer Systems, vol. 30, no. 4, November 2012.

[4] S. Che, J. Li, J. Lach, and K. Skadron. Accelerating compute intensive applications with GPUs and FPGAs. In Proc. of the 6th IEEE Symposium on Application Specific Processors, pp. 101-107, June 2008.

[5] D. Chisnall. The Definitive Guide to the Xen Hypervisor (Prentice Hall Open Source Software Development Series). Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[6] B. Cope, P. Y. K. Cheung, W. Luk, and S. Witt. Have GPUs made FPGAs redundant in the field of video processing?. In Proc. of the 2005 IEEE International Conference on Field-Programmable Technology, pp. 111-118, 2005.

[7] M. Dowty and J. Sugerman. GPU Virtualization on VMware's Hosted I/O Architecture. SIGOPS Operating Systems Review, vol. 43, no. 3, pp. 73-82, July 2009.

[8] E. El-Araby, I. Gonzalez, T. El-Ghazawi. Virtualizing and sharing reconfigurable resources in High-Performance Reconfigurable Computing systems. Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications, pp. 1-8, 2008.

[9] E. El-Araby, I. Gonzalez. Exploiting partial runtime reconfiguration for High-Performance Reconfigurable Computing. ACM Transactions on Reconfigurable Technology and Systems, vol. 1, no. 4, 2009.

[10] M. Frigo and S. G. Johnson. The design and implementation of fftw3.In Proc. of the IEEE, vol. 93, no. 2, pp. 216-231, February 2005.

[11] G. Giunta, R. Montella, G. Agrillo and G. Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In Proc. of Euro-Par, pp. 379-391, Heidelberg, 2010.

[12] R. P. Goldberg. Architectural principles for virtual computer systems. Ph.D. Thesis. Division of Engineering and Applied Physics, Harvard University Cambridge Massachusetts, 1973.

[13] I. Gonzalez, S. Lopez-Buedo, G. Sutter, D. Sanchez-Roman, F. J. Gomez-Arribas, J. Aracil. Virtualization of reconfigurable coprocessors in HPRC systems with multicore architecture. Journal of Systems Architecture, vol. 58, no. 6, pp. 247-256, March 2012.

[14] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: GPU-Accelerated Virtual Machines. In Proc. of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing, pp. 17-24, 2009.

[15] C. H. Huang, P. A. Hsiung, J. S. Shen. Model-based platform-specific co-design methodology for dynamically partially reconfigurable systems with hardware virtualization and preemption. Journal of Systems Architecture, vol. 56, no. 11, pp. 545-560, August 2010.

[16] W. Huang, M. Koop, Q. Gao, and D. K. Panda. Virtual machine aware communication libraries for high performance computing. In Proc. of the ACM/IEEE conference on Supercomputing, November 2007.

[17] K. Kim, C. Kim, S.-I. Jung, H. Shin and J.-S. Kim. Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen. In Proc. of the 4th International Conference on Virtual Execution Environments, pp. 11-20, 2008.

[18] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: The Linux Virtual Machine Monitor. In Proc. of the Linux Symposium, pp. 225-230, 2007.

[19] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara. VMM-independent graphics acceleration. In Proc. of the 3rd International Conference on Virtual Execution Environments, pp. 33-43, June 2007.

[20] E. Lübbers. Multithreaded Programming and Execution Models for Reconfigurable Hardware. PhD thesis, Computer Science Department, University of Paderborn, 2010.

[21] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling I/O in virtual machine monitors. In Proc. of the 4th International Conference on Virtual Execution Environments, pp. 1-10, 2008.

[22] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In Proc. of the 20th international symposium on High performance distributed computing, pp. 217-228, 2011.

[23] M. Sabeghi and K. Bertels. Toward a runtime system for reconfigurable computers: A virtualization approach. Design, Automation & Test in Europe Conference & Exhibition, pp. 1576-1579, April 2009.

[24] L. Shi, H. Chen, and J. Sun. vCUDA: GPU Accelerated High Performance Computing in Virtual Machines. IEEE Transactions on Computers, vol. 61, no. 6, pp. 804-816, June 2012.

[25] J. Wang, K.-L. Wright, and K. Gopalan. XenLoop: A transparent high performance inter-VM network loopback. In Proc. of the 17th International Symposium on High Performance Distributed Computing, pp. 109-118, June 2008.

[26] J. Wiltgen and J. Ayer.Bus Master Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions. Xilinx Corporation, September 2011.

[27] C. Young, J. A. Bank, R. O. Dror, J. P. Grossman, J. K. Salmon, and D. E. Shaw. A 32×32×32, spatially distributed 3D FFT in four microseconds on Anton. In Proc. of the Conference on High Performance Computing Networking, Storage and Analysis, 2009.

[28] X. Zhang, S. McIntosh, P. Rohatgi, and J. Griffin. XenSocket: A high-throughput interdomain transport for VMs. In Proc. of the 2007 International Conference on Middleware, pp.184-203, 2007.

[29] LogiCORE IP Fast Fourier Transform v7.1.Xilinx Corporation, March 2011.

[30] Virtex-6 FPGA Integrated Block for PCI Express, Xilinx Corporation, September 2010.

[31] Virtex-6 FPGA Memory Interface Solutions, Xilinx Corporation. June 2011.

[32] DMA Back-End Core User Guide., Northwest Logic Corporation, 2010.

[33] Xen PCI Passthrough, http://wiki.xen.org/wiki/Xen_PCI_Passthrough.