pvFPGA: paravirtualising an FPGA-based hardware accelerator towards general purpose computing

Wei Wang, Miodrag Bolic* and Jonathan Parri

Computer Architecture Research Group,
University of Ottawa,
800 King Edward, Ottawa, Ontario, Canada
Email: weiwang.carg@yahoo.com
Email: mbolic@eecs.uottawa.ca
Email: jparri@uottawa.ca
*Corresponding author

Abstract: This paper presents an ameliorated design of pvFPGA, which is a novel system design solution for virtualising an FPGA-based hardware accelerator by a virtual machine monitor (VMM). The accelerator design on the FPGA can be used for accelerating various applications, regardless of the application computation latencies. In the implementation, we adopt the Xen VMM to build a paravirtualised environment, and a Xilinx Virtex-6 as an FPGA accelerator. The data transferred between the x86 server and the FPGA accelerator through direct memory access (DMA), and a streaming pipeline technique is adopted to improve the efficiency of data transfer. Several solutions to solve streaming pipeline hazards are discussed in this paper. In addition, we propose a technique, hyper-requesting, which enables portions of two requests bidding to different accelerator applications to be processed on the FPGA accelerator simultaneously through DMA context switches, to achieve request level parallelism. The experimental results show that hyper-requesting reduces request turnaround time by up to 80%.

Keywords: field-programmable gate array; FPGA; hardware accelerator; virtualisation; hyper-requesting; streaming pipeline; DMA context switch.

Reference to this paper should be made as follows: Wang, W., Bolic, M. and Parri, J. (2017) 'pvFPGA: paravirtualising an FPGA-based hardware accelerator towards general purpose computing', *Int. J. High Performance Computing and Networking*, Vol. 10, No. 3, pp.179–193.

Biographical notes: Wei Wang received his BEng in Electronics and Information Engineering from the Changzhou Institute of Technology, China in 2011, and MASc in Electrical and Computer Engineering from the University of Ottawa, Canada in 2013. He was a Research Assistant at the Computer Architecture Research Group at the University of Ottawa. His research interests include computer architecture, virtualisation technology, and hardware/software co-design. He is currently a Virtualisation Software Developer at Intel.

Miodrag Bolic received his BSc and MSc in Electrical Engineering from the University of Belgrade, Serbia in 1996 and 2001, and PhD in Electrical Engineering from Stony Brook University, USA in 2004. From 1996 to 2000, he was a Research Associate with the Institute of Nuclear Sciences, Vinča, Belgrade, Serbia. Since 2004, he has been with the University of Ottawa, Canada where he is an Associate Professor with the School of Electrical Engineering and Computer Science. His research includes hardware/software accelerators, biomedical signal processing and RFID. He published about 40 journal papers, four book chapters and edited one book. He currently serves as an Associate Editor for the *International Journal of Reconfigurable Computing*, Hindawi and has been involved in organisation of a number of conferences.

Jonathan Parri is a PhD candidate at the University of Ottawa and a senior member of the Computer Architecture Research Group. His current research focuses on design space exploration in the hardware/software domain, targeting embedded and traditional systems. He has been employed in many capacities from developer and technical writer to research engineer.

This paper is a revised and expanded version of a paper entitled 'pvFPGA: accessing an FPGA-based hardware accelerator in a paravirtualized environment' presented at CODES+ISSS 2013, Montreal, Canada, 29 September to 4 October 2013.

1 Introduction

Cloud computing services offer many advantages for potential customers: low start-up cost, high-availability, instant access to massive computing power, no need for in-house technical expertise, and so on (Eguro and Venkatesan, 2012). Virtualisation technology, a key component of cloud computing, has generated great interest recently. With virtualisation, applications running on different operating systems (OS) can access the same hardware resources. This is achieved by allowing the underlying hardware resources to be shared by multiple virtual machines (VMs) or domains (Xen-specific term of VMs) with each running a separate OS. The virtual machine monitor (VMM) is responsible for separating each running instance of an OS from the physical machine, and guarantee that these OSs do not interfere with one another. The general benefits of virtualisation are:

1 creating higher hardware utilisation

2 reducing the number of hardware machines thereby reducing financial costs and power consumption

3 facilitating OS migration.

x86 machines are historically difficult to virtualise, because some sensitive instructions of x86 (e.g., SIDT) do not trap when executed in user mode. Full virtualisation, including hardware-assisted virtualisation [e.g., Intel VT-x (Neiger et al., 2006)], and paravirtualisation are the two prevailing virtualisation solutions for x86 machines. VMware workstation (Bugnion et al., 2012) and KVM (Kivity et al., 2007) are the two well-known VMMs that support full virtualisation. The bedrock of full virtualisation is trap-and-emulate, which traps the privileged instructions to their emulated versions to implement privileged operations during runtime. This keeps the virtualisation layer transparent to OSs, but it leads to significant performance penalties owing to the dynamic trap-and-emulate overhead (Walters et al., 2008). The Xen VMM (Barham et al., 2003) is famous for its paravirtualisation support. Paravirtualisation requires OS modification to support paravirtualisation, but it has high efficiency in performing I/O operations (Chisnall, 2007). A common hybrid solution is to use paravirtualised drivers in full virtualisation to achieve high I/O performance.

Managing and analysing large and complex datasets have recently brought a big challenge to the cloud infrastructure (Agrawal et al., 2011). We view the popular solutions like Google MapReduce (Dean and Ghemawat, 2008) and Hadoop (http://hadoop.apache.org/) as horizontal solutions, which lay on the foundation of building large clusters with adding more nodes. Vertical solutions (e.g., Canny and Zhao, 2013) aim at enhancing single node compute capability. Horizontal and vertical solutions are not mutually exclusive, that is, they can be combined en masse to deal with the 'big data' in the cloud.

Single node computing enhancement can be achieved through adding hardware accelerators to hardware servers. Hardware accelerators assist central processing units (CPUs) in speeding up computations of complex algorithms in various fields, such as video/image processing (Yamaoka et al., 2006; Jan et al., 2015; Taibo et al., 2011), signal processing (Lee et al., 2013) and various mathematical calculations (Tian and Benkrid, 2010; Okuyama et al., 2012; Rostrup et al., 2013). Graphics processing units (GPUs) and field-programmable gate arrays (FPGAs) are two types of dominantly used hardware accelerators (Chung et al., 2010). GPUs are inexpensive (Brodtkorb et al., 2013, and popular for general purpose computing (Owens et al., 2007). Since GPUs are programmed using high level languages and APIs which abstract away hardware details (Che et al., 2008), they are suitable to be used by software developers. FPGAs are highly customisable and reconfigurable, and achieve highly efficient algorithm acceleration with deep pipelining and parallelism (register level). Partial runtime reconfigurability is another important distinguishing feature of FPGAs (Silva and Ferreira, 2006). FPGAs have been found to outperform GPUs in many specific applications (Che et al., 2008; Cope et al., 2005). Since 2007, many researchers (Dowty and Sugerman, 2009; Gupta et al., 2009; Shi et al., 2012; Giunta et al., 2010; Ravi et al., 2011; Lagar-Cavilla et al., 2007) have been focusing on making GPUs a shared resource within a virtualised environment, which would allow for adding GPUs to the infrastructure level of cloud computing. But the idea of adding FPGA accelerators to cloud computing (El-Araby et al., 2008; Gonzalez et al., 2012; Huang et al., 2010; Huang and Hsiung, 2013; Lübbers, 2010; Sabeghi and Bertels, 2009; Jain et al., 2014; Byma et al., 2014; Wang et al., 2013) still stays at an exploration stage.

We propose pvFPGA, a leading edge heterogeneous system design solution, which efficiently virtualises an FPGA-based hardware accelerator by a VMM on the x86 platform. The benefits of pvFPGA are the combinations of the benefits of virtualisation and the benefits of using FPGA accelerators, which are:

1 the utilisation of an FPGA accelerator is increased

2 applications running in different domains can obtain speedup in the computation of complex algorithms using an FPGA accelerator.

Service is the theme of cloud computing. Different qualities of service (QoS) are offered to cloud clients according to different tiers. Similarly, pvFPGA should be able to supply guest domains with different maximum data transfer bandwidths so that higher tier clients will complete their acceleration requests faster than ordinary clients.

Preliminary results of the project were presented in Wang et al. (2013), while an upgraded and more advanced design of pvFPGA is detailed in this paper. Basic overview of the Xen VMM is given in Wang et al. (2013), and will not be repeated it in this paper. But a general understanding of the Xen VMM is recommended to understand this work. The experimental results in Wang et al. (2013) show that:

1 the virtualisation overhead in pvFPGA is near zero when the allocated data pool size is larger than 4 MB

2 the coprovisor successfully multiplexes acceleration requests from multiple domains

3 different QoS levels are assigned to domains by assigning them different data transfer bandwidths.

In comparison to Wang et al. (2013), the main contributions of this paper are summarised as follows:

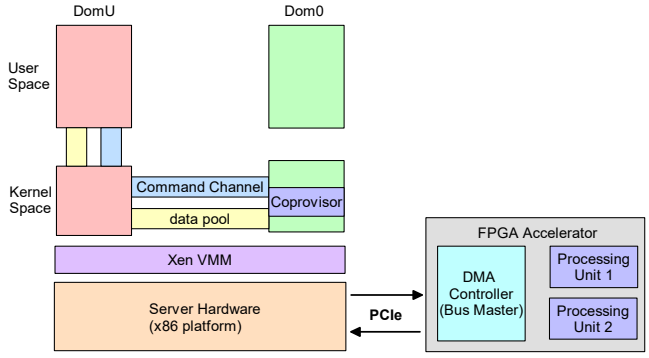
- the accelerator design supports more than one accelerator application on the FPGA which can be requested by user applications from domains
- the coprovisor adopts the streaming pipeline technique that facilitates general purpose computing
- the coprovisor supports hyper-requesting, which enables requests destined to different accelerator applications to be processed simultaneously, thereby achieving request level parallelism.

The remainder of this paper is organised as follows. We begin in Section 2 with an elaboration of our pvFPGA design. In Section 3, we make a detailed analysis of the streaming pipeline technique that can be used in an FPGA accelerator design. Section 4 shows the implementation and evaluation of pvFPGA, as well as detailed analyses of the experimental results. In Section 5, we compare pvFPGA with recent accelerator virtualisation solutions. Ultimately, we conclude this paper and discuss possible future directions in Section 6.

2 System design

Dom0 and DomU are the two types of domains supported by Xen. Dom0 is the privileged domain, which can access external devices directly. DomU represents an unprivileged domain, which requires the use of a split device driver model to access devices via Dom0.

Figure 1 An overview of pvFPGA framework (see online version for colours)



An overview of pvFPGA is shown in Figure 1. On the hardware side, we use a DMA controller (Northwest Logic Corporation, 2010) on the FPGA accelerator for data transfer between the server memory and the FPGA accelerator. Our current implementation includes two accelerator applications (processing units), this can be easily extended to three or more accelerators. On the software side, pvFPGA includes three main components: a data pool, a command channel and a coprovisor. A data pool consists of a group of physical memory pages, which are allocated in a DomU kernel space and shared with the Dom0 kernel space through the grant table mechanism supported by the Xen VMM. A user process in a DomU can map the data pool into its address space so that it can directly store data into the data pool. Thus, the data pool is used for user-kernel and inter-domain data transfer. Finally, the data pool is exposed to the DMA controller on the FPGA accelerator as a DMA buffer for data fetching and results writing back. Similar to a data pool, a command channel is a shared memory-based channel for command information transfer. Our design currently uses the command channel to transfer two command elements:

1 the application number that the DomU requires to use on the FPGA accelerator

2 the size of the data in the data pool that must be transferred to the FPGA accelerator.

For example, a DomU has a 4 MB data pool, but it may request only 1 MB data to be transferred for processing on the FPGA accelerator. A coprovisor is a component that we propose for multiplexing requests from DomUs to access the shared FPGA accelerator.

Table 1 Descriptions of the operation flow

Steps	*Descriptions*
1	An application specifies the app number and data size in the command channel, which is mapped to its address space through the 'mmap( )' call.
2	The application directly puts data in the shared data pool, which 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 a system call, e.g., ioctl).
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.

Both the data pool and command channel in a DomU are exposed as device nodes in the */dev* directory. We wrote some system call implementations, and user processes can interact with them through the system calls. Table 1 shows an example of general implementation using pvFPGA. A more detailed introduction of the basic pvFPGA design is introduced in Wang et al. (2013). In this paper, we will introduce an upgraded design of pvFPGA.

2.1 FPGA accelerator design

Figure 2 shows the accelerator design on the FPGA. In the figure, same colour modules operate at the same frequency. The on-chip FIFO buffer is a memory queue IP core supplied by Xilinx for applications requiring in-order storage and retrieval (Xilinx Corporation, 2011b). The input and output of a FIFO buffer can operate at different frequencies. The PCIe interface (Xilinx Corporation, 2010) is adopted as the communication channel, and direct memory access (DMA) technique is used for efficient data transfer between the FPGA accelerator and the host server memory. For convenience in the paper discussion, 'DMA read operation' means that the DMA controller reads data from the host system's memory, and 'DMA write operation' means that the DMA controller writes data to the host system's memory.

As shown in Figure 2(a), we use two DMA channels to achieve full duplex; one for DMA read operations and one for DMA write operations. Thus, we can do DMA reads from the server memory and DMA writes to the server memory simultaneously; that is, we take advantage of a streaming pipeline. Figure 2(b) shows the pipeline stages. The overall procedure is performed in three stages, DMA read stage, computation stage, and DMA write stage. The latency for finishing each of the three stages is identical in Figure 2(b), but in practice, this is not always the case, since the latency of computation is determined by the accelerator application, and the latency of DMA read or write operations is determined by the communication channel. A detailed discussion of the pipeline technique will be shown in Section 3.

The app controller in Figure 2(a) performs four functions:

1 directing the input streaming data from the DMA controller to app1 or app2

2 multiplexing app1 and app2 to use the DMA write channel

3 maintaining the accelerator status word (ASW)

4 raising an interrupt when required.

The accelerator driver will direct the app controller to send the input block of data to either app1 or app2, by configuring the ASW before initiating the start of a DMA read operation. Also, the app controller needs to multiplex requests from app1 and app2 to use the DMA write channel. Since initiating a DMA write operation is also implemented by the accelerator driver, the app controller needs to maintain the ASW status, and ensures it remains visible to the driver.

Figure 2 Accelerator design on an FPGA, (a) accelerator design (b) pipeline stages (see online version for colours)

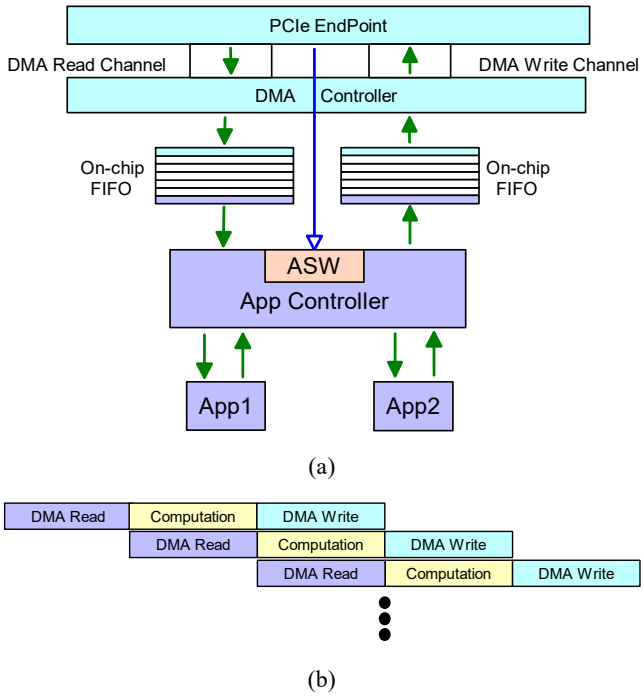


Table 2 Bits in the ASW

Bit	*Functions*
Bit 0	0-app1 is selected for the following block of data; 1-app2 is selected for the following block of data. Written by the driver.
Bit 1	This bit is set by the FPGA accelerator when one block of data finishes using the DMA read channel. An interrupt is raised after setting this bit. Read and Cleared by the driver.
Bit 2	0-app1 finishes its processing for one block of data; 1-app2 finishes its processing for one block of data. Read by the driver.
Bit 3	This bit is set by the FPGA accelerator either when app1 or app2 finishes a computation. An interrupt is raised after setting this bit. Read and cleared by the driver.
Bit 4–31	Reserved for future updates.

The introduction of the ASW enhances the interaction between the driver and the FPGA accelerator. More precisely, the driver gets to know the status of the FPGA accelerator through the ASW, and from the FPGA accelerator perspective, the ASW tells the FPGA accelerator what the driver needs. The ASW is mapped to the system memory through the PCIe base address register (BAR). The design of the ASW is shown in Table 2. With the help of the ASW, interrupts that are raised by the FPGA accelerator due to different events can be distinguished. When a block

of data finishes using the DMA read channel, the app controller will set bit 1 of the ASW and raise an interrupt; for convenience, we call this a type 1 interrupt. When an app (either app1 or app2) finishes a computation, the app controller will clear (if app1 finishes a computation) or set (if app2 finishes a computation) Bit 2 of the ASW, set bit 3 of the ASW, and raise an interrupt. We call this a type 2 interrupt in the later discussion. Thus, a type 1 interrupt is raised as a block of data finishes using the DMA read channel, and a type 2 interrupt is raised as a block of data finishes its computation on the FPGA accelerator.

2.2 FPGA virtualisation

2.2.1 Coprovisor

The Xen VMM scheduler is only responsible for controlling access to CPUs, while the backend drivers need to provide some means of regulating the number of I/O requests that a given domain can perform (Chisnall, 2007). To manage this for pvFPGA, we propose a component that we call a coprovisor, which multiplexes requests from different domains accessing the FPGA accelerator. For GPU virtualisation, the multiplexing work is performed in user space, owing to the lack of standard interfaces at the hardware level and driver layer. Precisely, the multiplexer and scheduler are put on the top of CUDA runtime or driver APIs (Gupta et al., 2009; Shi et al., 2012; Giunta et al., 2010). In our case, the coprovisor can perform multiplexing directly at the accelerator driver layer in Dom0.

The architecture of the coprovisor is shown in Figure 3. It consists of five parts: request inserter, scheduler, request remover and two request queues. Request queue 1 is used for buffering requests to access app1 on the FPGA accelerator, and request queue 2 is used for buffering requests to access app2. A DomU notifies the Dom0 via an event channel, so the request inserter, which is responsible for inserting requests from DomUs into the corresponding 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 corresponding request queue. The scheduler is responsible for scheduling requests from the two request queues to access the FPGA accelerator through the accelerator device driver. In terms of one request queue, requests are scheduled using first-come, first-served (FCFS) policy; that is, requests in the same queue are extracted in an orderly manner by the scheduler. More detail about the scheduling is discussed in Section 2.3.

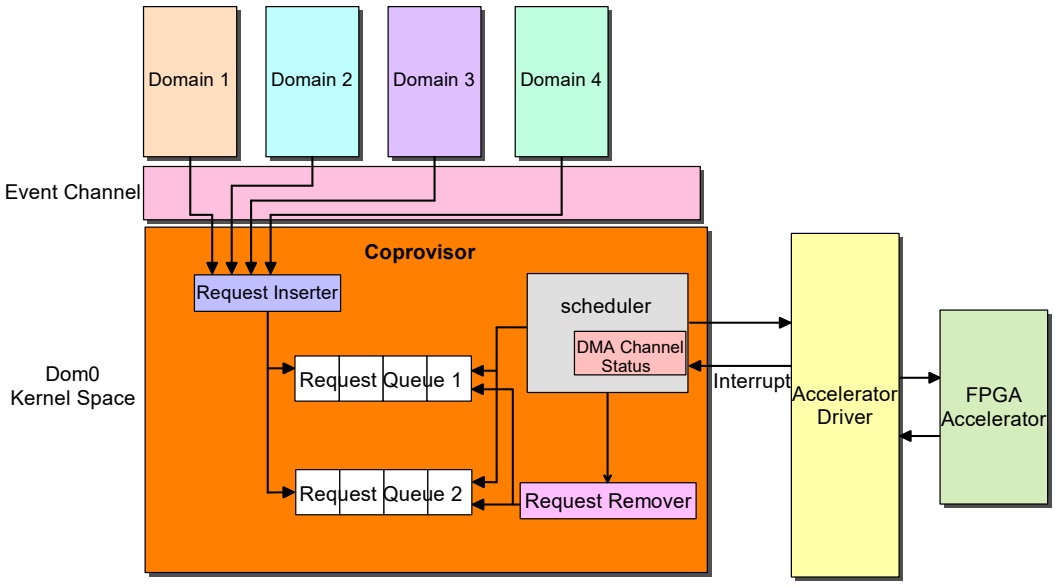
The size of a data pool in a DomU implies the maximum data transfer bandwidth. For instance, a DomU (D1) assigned with a 1 MB data pool can transfer 1 MB data at a maximum for each request to the FPGA accelerator, while a DomU (D2) assigned with a 512 KB data pool can transfer 512 KB data at a maximum per request. When the two DomUs contend for using the same app on the FPGA accelerator and the acceleration needs to send more than 512 KB data, D2 is slower because it needs to send more requests to complete the entire acceleration. To provide DomUs with different maximum data transfer bandwidths, the size of a data pool can be regulated in each DomU's frontend driver and the Dom0's backend driver.

2.3 Hyper-requesting

2.3.1 Case analysis

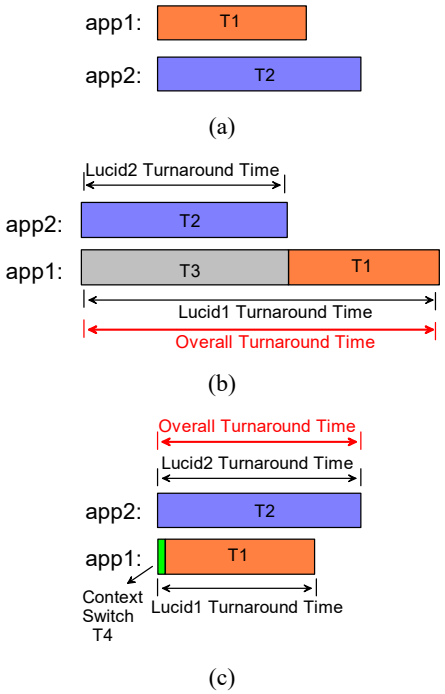
In this section, we propose the concept of hyper-requesting to improve request turnaround time through DMA context switches.

Figure 3 Design of a coprovisor (see online version for colours)



Let us consider two applications, app1 and app2, running on the FPGA accelerator as shown in Figure 4. The processing time needed by app1 and app2 for processing one block of data is $T1$ and $T2$, respectively [see Figure 4(a)]. Assume both $T1$ and $T2$ are longer than the communication latency of transferring one block of data. Two DomUs, Lucid1 and Lucid2, are requesting access to the FPGA accelerator simultaneously, with Lucid1 requesting app1 acceleration and Lucid2 requesting app2 acceleration. Suppose the Lucid2 request comes first and gets serviced before the Lucid1 request. Without hyper-requesting, the turnaround time of Lucid1 was $T1 + T3$ [see Figure 4(b)], where $T3$ equals $T2$. The Lucid1 request will not be scheduled to the FPGA accelerator until the Lucid2 request is serviced, which results in a scheduling delay ($T3$) for Lucid1 to get its request serviced. Even though the DMA read channel [in Figure 2(a)] becomes idle when a block of data is transferred to app2 for processing, the Lucid1 request has to wait to get serviced till app2 on the FPGA accelerator finishes its processing and raises an interrupt to the server. This unnecessary scheduling delay can be eliminated by implementing a DMA context switch. More precisely, once the Lucid2 request finishes its use of DMA read channel, an immediate context switch to the Lucid1 request will be implemented by the coprovisor.

Figure 4 Analysis of request turnaround time, (a) processing time of app1 and app2 (b) turnaround time without hyper-requesting (c) turnaround time with hyper-requesting (see online version for colours)



As shown in Figure 4(c), the overall turnaround time will be reduced to $T2$, and the turnaround time of Lucid1 is $T1$ plus the context switching overhead $T4$. The DMA context switching overhead is minor, because only one parameter (the bus address of the next DMA buffer descriptor) needs to be loaded to the DMA controller. Therefore, both Lucid1 and the overall turnaround time will be reduced through a DMA context switch. When two requests have multiple portions of data that are required to be processed, portions of the two requests can be loaded into the FPGA accelerator through DMA context switches to be processed simultaneously. The request turnaround time improvement will be shown in our experiments in Section 4.3.

2.3.2 Hyper-requesting design

In order to support hyper-requesting, requests to access the FPGA accelerator need to become context-aware so that request context switches can be performed. Similar to the functionality of a process control block (PCB), each request has its own request control block (RCB) which is used by the coprovisor to setup a DMA executing context. The design of an RCB is shown in Figure 5. Table 3 explains the fields of an RCB.

Figure 5 Design of a request control block

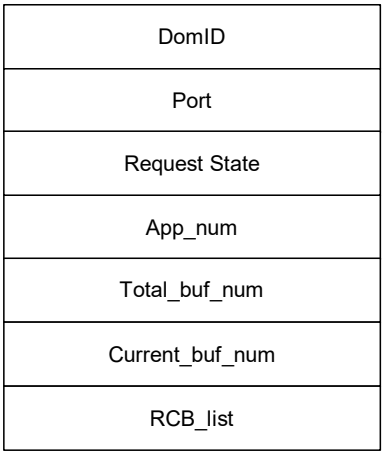


Table 3 Description of a request control block

Field	*Description*
DomID	Denotes the ID of the domain that the request comes from.
Port	The port number of the event channel. It is used to notify the request's domain through the corresponding event channel.
Request State	The three possible states of a request: DMAREAD, DMAWRITE and DMAFIN.
App_num	Specifies which app the request needs to use on the FPGA accelerator. 0-app1, 1-app2.
Total_buf_num	The total number of buffer fragments used by the request.The size of one DMA buffer fragment is 4KB in our implementation.
Current_buf_num	Specifies the number of current buffer fragment that needs to be transferred to the FPGA accelerator.
RCB_list	A pointer to the next request.

The scheduler in Figure 3 mainly performs four functions:

1 Exposing the status of both DMA read and DMA write channels to the device driver (such as if the DMA read channel is idle or in use).

2 Scheduling a request to use the DMA read channel (this may need to implement a DMA context switch).

3 Maintaining the RCBs of the head requests of the two queues; for example, when one buffer fragment (storing one block of data) of a request is scheduled to use the DMA read channel, the request state will be updated with DMAWRITE, and the 'current_buf_num' will be incremented by one.

4 Invoking the request remover to remove a request once the request has completed using the FPGA accelerator.

Figure 6 shows the request state transition diagram, and the DMA read channel state transition diagram is shown in Figure 7. When a request is received from a DomU it is marked with DMAREAD state, which indicates that the request is waiting for a DMA read operation. If the DMA read channel is in IDLE state, the DMA read operation will be initiated, the DMA read channel state will be updated to BUSY, and the request state will be updated to DMAWRITE. The scheduler is invoked in the following three cases:

1 a new request is inserted into a queue

2 a type 1 interrupt is received

3 after initiating a DMA write operation.

When the scheduler is invoked, it will check if the DMA read channel is idle and a head request in one of the queues is in DMAREAD state. If so, it will schedule the head request for a DMA read operation.

Figure 6 Request state transition diagram

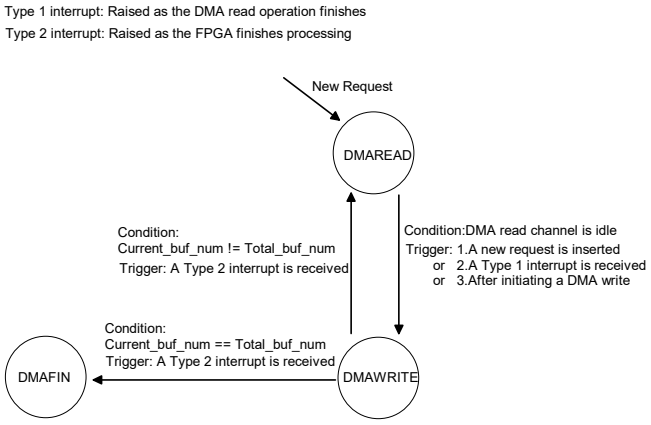
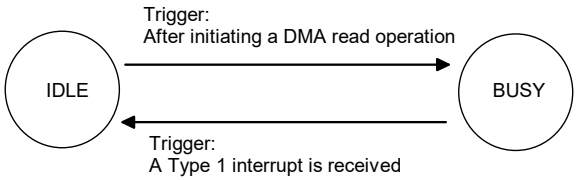


Figure 7 DMA read channel state transition diagram



When a DMA read operation finishes, a type 1 interrupt will be received from the FPGA accelerator to release the DMA read channel, thereby modifying the DMA read channel state to IDLE. In this case, if the head request in the other queue is waiting for a DMA read operation, the operation can be initiated. When a type 2 interrupt is received, the context of a request with DMAWRITE state will be loaded into the DMA controller to implement a DMA write operation, and the 'Current_buf_num' will be incremented by one. If the head requests of the two queues are both in DMAWRITE state, the bit 2 of the ASW informs the Scheduler which request has finished a computation on the FPGA.

After initiating a DMA write operation there are two possible cases. One case is that the request has not been completely serviced; that is, there are still data in the DMA buffer associated with the request that have not been processed. In this case, the state of this request will be updated to DMAREAD after initiating the DMA write operation. The second case is that a request has finished all its data processing (Current_buf_num == Total_buf_num). The request will be set to the DMAFIN state, and the request remover will be invoked to remove the request from the queue.

3 Streaming pipeline analysis

With the pipeline technique, some execution latency can be hidden. For example, in the third cycle in Figure 2(b), the DMA read and DMA write operations are implemented simultaneously as the FPGA accelerator is doing a computation, so the latency of the DMA read and write operations is hidden. In the following discussion, we assume the computation latency is T_C, and the DMA data transfer latency of one block of data is T_D (DMA read latency = DMA write latency = T_D), and the total number of blocks of data is N.

The pipeline shown in Figure 2(b) is similar to a classic reduced instruction set computer (RISC) pipeline, where all the operations have identical execution latency. However, this type of pipeline seldom occurs with the design of an FPGA accelerator, because it is uncertain that the computation latency, TC, is equal to the DMA data transfer latency, TD. Following are two possible scenarios:

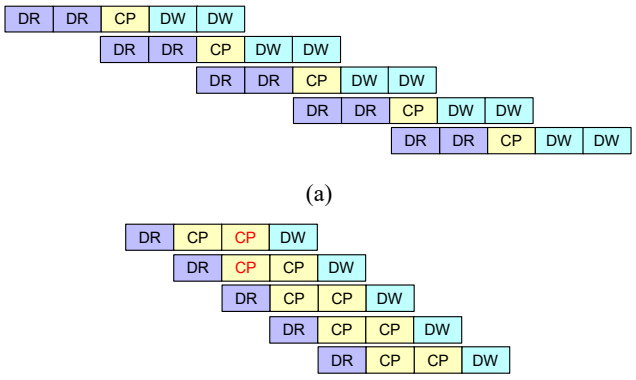
Scenario 1 In Figure 8(a), the DMA data transfer latency is larger than the FPGA accelerator computation latency, that is, $T_D > T_C$. One block in the figure occupies one timing unit. DR represents DMA read, and two concatenate DR blocks mean that the DMA read operation occupies two timing units; CP represents Computation on the FPGA; DW represents DMA write.

Scenario 2 In Figure 8(b), the DMA data transfer latency is less than the FPGA accelerator computation latency, that is $T_D < T_C$.

Pipelined operations in scenario 1 can work similarly to the RISC pipeline technique. However, the scenario 2 pipelined operations shown in Figure 8(b) will cause some problems. The overlapping of two computation operations [e.g., two red CPs in Figure 8(b)] implies that two blocks of data are contending for one computation module. In other words, when the second block of data arrives in the FPGA accelerator for doing a computation, the first block of data has not completed its computation. This will cause a conflict on using the computation module. We refer to problems with streaming data conflicting on using the computation module as *streaming pipeline hazards*.[1]

A simple solution to solve the streaming pipeline hazards shown in Figure 8(b) is to increase the operating frequency of the computation module, thereby reducing the computation latency, T_C. But this is not always feasible, because there is always a limit to the maximum frequency of the design. Exceeding the maximum frequency would cause issues such as setup/hold timing violations.

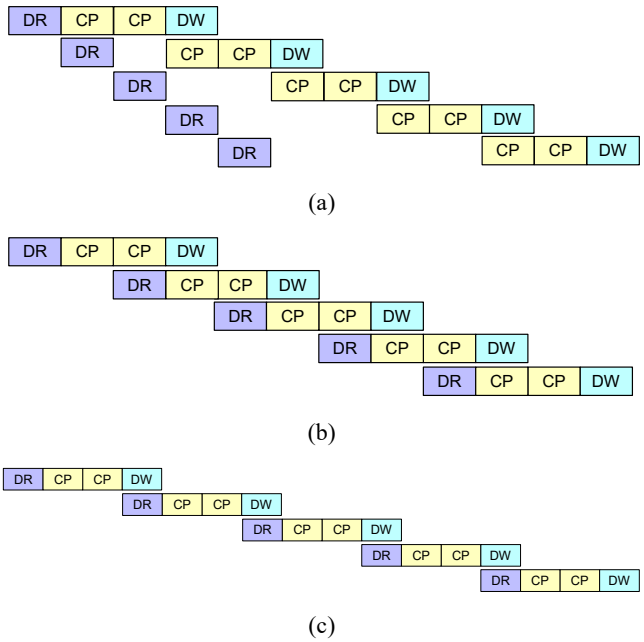
Figure 8 Examples of non-RISC-like pipeline stages, (a) an example of scenario 1 pipeline stages (b) an example of scenario 2 pipeline stages (see online version for colours)



The second solution to solve the streaming pipeline hazards is shown in Figure 9(a), where the start of next DMA read operation is delayed. This is very difficult to implement, because the time to start the DMA read operation needs to be at an accurate time instance when the FPGA is executing a computation. Figure 9(a) shows an example where T_C equals $2*T_D$, and the time to start the DMA read of next block of data can be set when the FPGA finishes the current half block of data computation. However, in practice T_C could be any factor larger than T_D (e.g., 1.5 or 1.75). Implementing this solution is complicated, and varies with different applications because they have different computation latencies.

The third solution is to delay the start of the computation module, as shown in Figure 9(b). It is clear in the figure that the second and third solutions have the same turnaround time. In practice, the solution shown in Figure 9(b) is prone to implement, because the start of next DMA read operation can be just triggered when the first DMA read operation finishes, but the incoming data are not processed immediately when they arrive in the FPGA accelerator. For example, when the second block of data is being computed, the third to fifth blocks of data have already arrived and must be buffered. It can be easily inferred from Figure 9(b) that the pipeline bubbles between DR and CP get increased when N increases. Thus, the disadvantage of the solution in Figure 9(b) is that it requires large memory to buffer the incoming data, and the required size of the buffer grows as N increases.

Figure 9 Solutions to solve streaming pipeline hazards, (a) delay the start of computation operations (b) delay the start of DMA read operations (c) simultaneous implementation of DMA read and write operations (see online version for colours)



The fourth solution is to add one more computation unit to the FPGA accelerator. In this case, the second block of data can be streamed to a different computation unit from the first block of data for a computation. However, implementing this solution has a similar difficulty to implementing the second solution. The example we show in Figure 9(b), where $T_C = 2*T_D$, needs two computation units in total. In practice, we may need to add more computation units on the FPGA based on the relationship between T_C and T_D. Thus, this solution cannot be adopted for general purpose use.

Figure 9(c) illustrates the fifth solution to address the problem; hide the DMA write operation latency by executing it simultaneously with a DMA read operation. The drawback of this solution is that it has longer turnaround time than the other solutions above. However, the fact that it can be used for general purpose computing is an advantage. That is, in practice it can be easily adopted for any situation regardless of the latency difference between the DMA data transfer and the FPGA computation. This type of pipeline satisfies our FPGA accelerator design aim, which is to make the FPGA accelerator capable of accelerating various applications for cloud clients. The

overall turnaround time of the pipeline shown in Figure 9(c) can be calculated by the following formula:

$$T_D + N * (T_C + T_D) \quad (1)$$

4 Implementation and evaluation

Table 4 shows the details of our experimental platform. We have verified the functionality and virtualisation overhead with a fast Fourier transform (FFT) benchmark, which generates the same results as those shown in Sabeghi and Bertels (2009). Thus, we will not repeat the results in this paper. In Section 4.1, we will measure the DMA data transfer latency, T_D, with a loopback application, and the result of T_D will be used for the following analyses. In Section 4.2, we will verify the functionality of the coprovisor for multiplexing requests to the same acceleration application and make a detailed analysis of the contention. Finally, we will evaluate the improvement in request turnaround time with hyper-requesting in Section 4.3.

Table 4 Experimental platform

Name	*Description*
x86 server	Intel® Xeon Processor W3670 running at 3.2 GHZ and 4 GB of main memory
FPGA accelerator	Virtex-6 LXT FPGA ML605 Evaluation Kit, 4-lane (Gen 2) PCIe interface
VMM	Xen-4.1.2
Native Linux without the Xen VMM	Ubuntu 12.04LTS
Dom0	Ubuntu 12.04LTS
4 DomUs	Ubuntu 10.04LTS, named with Lucid1, Lucid2, Lucid3, Lucid4, respectively

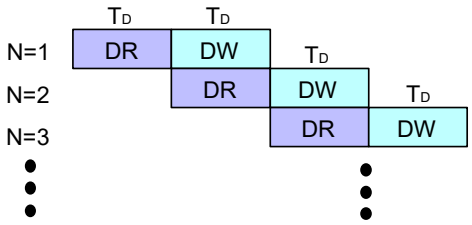
4.1 DMA data transfer latency measurement

In order to measure the DMA data transfer latency, we use a loopback application, which simply returns any unmodified data it receives, on the FPGA accelerator in the position of app1 in Figure 2(a). The DMA read and write operations are implemented with 4 KB data size as a block. The FPGA computation latency, T_C, can be considered zero, since there is no computation of the loopback application. The pipeline of the loopback application is depicted in Figure 10. When T_C equals zero, formula (1) can be simplified as:

$$T_D + N * T_D \quad (2)$$

The formula (2) shows the turnaround time of the loopback application. When large amounts of data (e.g., N = 1,024) are transferred to the loopback benchmark, we can get an accurate value of T_D, which is the DMA data transfer latency of one block of data. When 4 MB data (N = 1,024) is sent to the FPGA, the turnaround time is 3,586 us. According to formula (2), we get T_D equal to 3.5 us, which is the time of transferring 4 KB data. This value will be used for analyses of the following experiments.

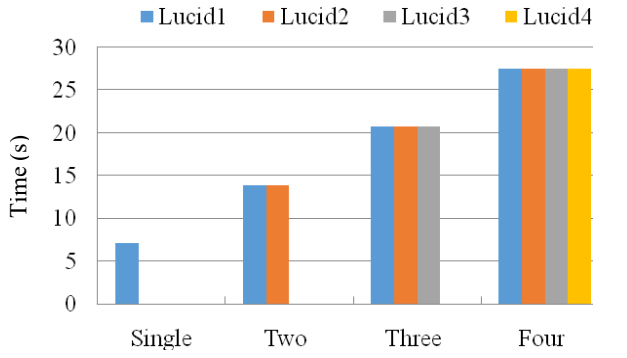
Figure 10 Pipeline stages of the loopback application (see online version for colours)



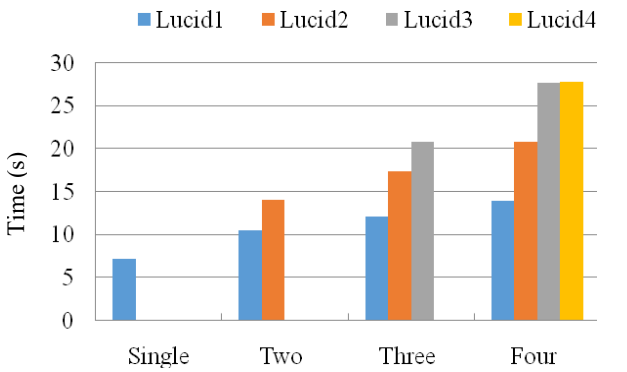
4.2 Coprovisor evaluation

The evaluation was presented in our previous paper (Wang et al., 2013). Since the coprovisor was significantly modified, we will re-evaluate the functionality of the coprovisor in this section and give a detailed contention analysis to show validity of experimental results in Section 4.2.1.

Figure 11 Evaluation of the coprovisor, (a) identical data pool size in DomUs (b) different data pool sizes in DomUs (see online version for colours)



(a)



(b)

In the experiments, we built a benchmark based on Xilinx FFT IP core (Xilinx Corporation, 2011a) for accelerating 256-point floating point (single precision) FFT. Executing this FFT benchmark requires at least 2 KB data (1 KB real

number and 1 KB complex number) to be transferred to the FPGA accelerator. Thus, each block of data has two FFT computations (4 KB/2 KB = 2). The FFT benchmark module operates at 250 MHz. One block of data in our implementation is 4 KB, so the FFT benchmark performs two FFT computations each time. We obtain the FFT benchmark computation latency (approximately 9.5 us) through a simulation in Modelsim, that is, T_C equals 9.5 us.

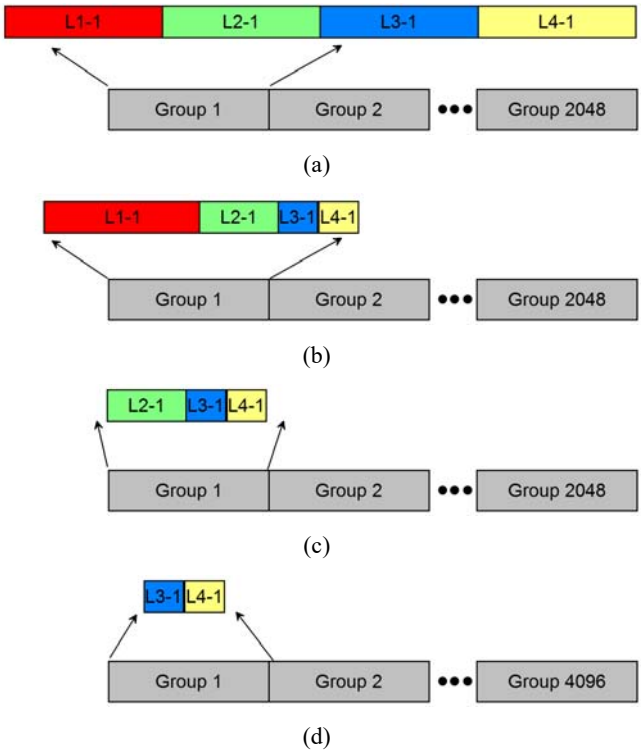
In the first evaluation, each of the four DomUs (Lucid1, Lucid2, Lucid3 and Lucid4) allocates a 1MB size data pool, which is shared with Dom0 for inter-domain data transfer. The FFT benchmark is used as app1 [see Figure 2(a)] for the evaluation. We set all the applications in the four DomUs to select app1 when they request services, and the data size for all the requests is set to 1MB via the command channel. In this evaluation, each of the DomUs sends 2 GB of data to compute FFT on the FPGA accelerator, so each DomU needs to send 2,048 requests to complete the computations. In Figure 11(a), when all four DomUs contend for one FPGA accelerator, all the DomUs spend equal time to complete the computations.

In the second evaluation, we assigned Lucid1, Lucid2, Lucid3 and Lucid4 with 1 MB, 512 KB, 256 KB and 256 KB data pool sizes respectively, separately in their kernel spaces. In this case, Lucid1 is treated as a higher tier domain, as it is assigned the larger data pool size, which implies higher maximum data transfer bandwidth. We set all the applications in the four DomUs to set the data size for all the requests equal to the data pool size via the command channel. In Figure 11(b), when all four DomUs request access to one FPGA accelerator simultaneously, Lucid1 completes the evaluation task twice faster than Lucid3, and Lucid3 and Lucid4 spend equal time to complete their evaluation tasks, due to their equal assigned data pool size. The request contention analysis of the two cases are elaborated in the following subsection.

4.2.1 Contention analysis

We first consider the situation in Figure 11(a) where the four DomUs contend for access to the FPGA accelerator with equal sized data pools. Figure 12(a) shows how these requests are scheduled. L1-1 represents the first request from Lucid1, L2-1 represents the first request from Lucid2, and so on. In Figure 11(a), each DomU is assigned a 1 MB data pool, so each domain needs to send 2,048 requests to complete 2 GB data of FFT computations. A DomU cannot send the second request until the first request is serviced, because each DomU has only one data pool. Thus, all the requests can be grouped into 2,048 groups, with each group having a request from each of the DomUs. Here, every request has 256 blocks of data (1 MB/4 KB); that is, N equals 256. After sending 2048 groups of requests, all the DomUs will complete their tasks. They finish with almost the same turnaround time, which is also the turnaround time of the 2048 groups of requests. One group of requests has four requests from each of the four DomUs. According to formula (1), the turnaround time of one group of requests, denoted as *Tg*, is [3.5 + 256 * (9.5 + 3.5)] * 4 = 13,326 us. Thus, the turnaround time of 2,048 groups of requests is 2,048 * *Tg* = 27.3 seconds, which means each of the four DomUs needs 27.3 seconds to get their requests serviced. Before sending a request, each DomU needs to load data into the data pool; this can be viewed as preprocessing work. However, it is not necessary to add the preprocessing time to the calculated result. In Figure 12(a), when the first request from Lucid1 (L1-1) finishes, it will load new data to the data pool for the second request (L1-2 in Group 2). Meanwhile, the first request from Lucid2 (L2-1) is scheduled for computations. The preprocessing time of L1-2 overlaps the time for servicing L2-1, so the preprocessing time is hidden. The experimental result in Figure 11(a) is 27.5 us which is very close to the calculated result.

Figure 12 Analysis of four DomUs contending for access to the shared FPGA accelerator,
(a) contention analysis of Figure 11(a)
(b) Phase 1 contention analysis of Figure 11(b)
(c) Phase 2 contention analysis of Figure 11(b)
(d) Phase 3 contention analysis of Figure 11(b)
(see online version for colours)



We next consider the situation in Figure 11(b), where the four DomUs that are assigned different data pool sizes contend for access to the shared FPGA accelerator. The contention analysis is more complex, since higher tier domains will complete their requests first and are out of the contention at a certain time instance. Here, we split the contention into three phases. In each phase a DomU completes its evaluation task, and will be out of contention for access to the shared FPGA accelerator. Phase 1 has Lucid1, Lucid2, Lucid3 and Lucid4 contending for the FPGA accelerator, while in Phase 2 only Lucid2, Lucid3 and Lucid4 contend for the accelerator, since Lucid1 has completed all its requests and is out of contention at the end

of Phase 1. In Phase 3, Lucid3 and Lucid4 contend for the FPGA accelerator, since Lucid2 has finished all its requests and is out of contention at the end of Phase 2.

Phase 1 Lucid1, Lucid2, Lucid3 and Lucid4 contend for access to the shared FPGA accelerator. Lucid1 will complete all its requests at the end of Phase 1. Figure 12(b) shows how the requests are scheduled, which is slightly different than that in Figure 12(a). L1-1 is twice as long as L2-1 and four times longer than L3-1, because Lucid1 is assigned a 1 MB data pool, while Lucid2 is assigned a 512 KB data pool and Lucid3 a 256 KB data pool. After 2,048 groups of requests, Phase 1 ends because Lucid1 has completed its task (1 MB * 2048 = 2 GB). After Phase 1, Lucid2, Lucid3 and Lucid4 have only partially finished their tasks. Table 5 shows the remaining data size of each DomU at the end of Phase 1.

In Phase 1, the turnaround time of one group of requests, $Tg1$, is equal to the sum of the request turnaround time of the four DomUs. Therefore, according to (1):

$$\begin{aligned} Tg1 &= 3.5 + 256*(9.5+3.5) + 3.5 + 128*(9.5+3.5) \\ &\quad + 3.5 + 64*(9.5+3.5) + 3.5 + 64*(9.5+3.5) \\ &= 6,670us. \end{aligned}$$

The turnaround time of Lucid1 equals the 13.7 seconds (2,048 * $Tg1$) turnaround time of Phase 1. As shown in Figure 11(b), our experimental result of 13.9 seconds is close to the calculated result.

Phase 2 As shown in Figure 12(c), in this phase Lucid2, Lucid3 and Lucid4, contend for access to the shared FPGA accelerator. Table 4 shows that Lucid2 has 1 GB data left for FFT computations, so the number of groups in Phase 2 is 2,048 (1 GB/512 KB). Table 6 shows the remaining data size of each DomU at the end of Phase 2.

In Phase 2, the turnaround time of one group of requests, $Tg2$, equals the sum of the request turnaround time of Lucid2, Lucid3 and Lucid4. Therefore:

$$\begin{aligned} Tg2 &= 3.5 + 128*(9.5+3.5) + 3.5 \\ &\quad + 64*(9.5+3.5) + 3.5 + 64*(9.5+3.5) \\ &= 3,338.5\,us. \end{aligned}$$

The turnaround time of Lucid2 is equal to the turnaround time of Phase 1 and Phase 2, which is 13.7 s + 2,048 * 3,338.5 us = 20.5 seconds. Our experimental result is 20.8 seconds which is close to the concluded result.

Phase 3 As shown in Figure 12(d), only Lucid3 and Lucid4 contend for access to the shared FPGA accelerator. So we get:

$$\begin{aligned} Tg3 &= 3.5 + 64*(9.5+3.5) + 3.5 + 64*(9.5+3.5) \\ &= 1,671us. \end{aligned}$$

Both Lucid3 and Lucid4 have 1GB data left for computations, so after 4,096 groups (1 GB/256 KB) of requests are scheduled Lucid3 and Lucid4 have completed their tasks. Therefore, the turnaround time of Lucid3 equals the turnaround time of Lucid4, which is also the turnaround time of the three phases. This is calculated as: 20.5 s + 4,096 * 1,671 us = 27.3 seconds. Our experimental result is 27.7 seconds, which is close to the calculated result.

As shown, the overall turnaround time (Phase 1 + Phase 2 + Phase 3) of Figure 11(b) equals that of Figure 11(a), since their overall data sizes for FFT computations are the same (4 * 2 GB = 8 GB). In Figure 11(a), all the DomUs get their requests serviced at an equivalent and invariable speed. In Figure 11(b), Lucid3 and Lucid4 get their requests serviced at a slower speed at the beginning, but as Lucid1 and Lucid2 drop out of contention the speed increases.

Table 5 Remaining data size of each DomU at the end Phase 1

DomU	*Remaining data size*
Lucid1	2 GB–1 MB * 2,028 = 0
Lucid2	2 GB–512 KB * 2,048 = 1 GB
Lucid3	2 GB–256 KB * 2,048 = 1.5 GB
Lucid4	2 GB–256 KB * 2,048 = 1.5 GB

Table 6 Remaining data size of each DomU at the end Phase 2

DomU	*Remaining data size*
Lucid2	1 GB–512 KB * 2,048 = 0
Lucid3	1.5 GB–256 KB * 2,048 = 1 GB
Lucid4	1.5 GB–256 KB * 2,048 = 1 GB

4.3 Hyper-requesting evaluation

In this section, we evaluate the improvement of request turnaround time due to hyper-requesting. The aim of proposing hyper-requesting is to enable two requests bidding to different accelerator applications to be processed simultaneously on the FPGA accelerator through DMA context switches, thereby reducing the request turnaround time. For convenience, we refer to the design presented in Wang et al. (2013), which does not have the ability to perform hyper-requesting as a basic design, and the design presented in this paper, which is able to perform hyper-requesting, as an improved design. All the above evaluations were implemented with requests requesting to use the same accelerator application (app1) on the FPGA accelerator, so hyper-requesting was not used. To evaluate

the request turnaround time improved by hyper-requesting, requests from Lucid1 and Lucid2 are set to access app1, and requests from Lucid3 and Lucid4 are set to access app2.

At the FPGA accelerator end, we used a design that emulates the computation procedure. As shown in Figure 13, both app1 and app2 have a buffer for storing a block of incoming data, and a timer for emulating algorithm computation time. The timer asserts an interrupt request to the app controller when the time is up. From the acceleration module perspective, the major difference between different accelerator applications involves varying computational latencies. The previous experiments are implemented with an app whose computation latency is at the microseconds level. In order to show that our proposed accelerator design can be used for general purpose computing, regardless of the app computation latency, we set the app1 timer to 4 seconds and the app2 timer to 2 seconds for one block of data (4 KB in the experiments). In this case, app1 and app2 emulate two algorithm computations that require 4 seconds and 2 seconds respectively to complete one block of data processing.

Figure 13 An accelerator emulating design for verification purposes (see online version for colours)

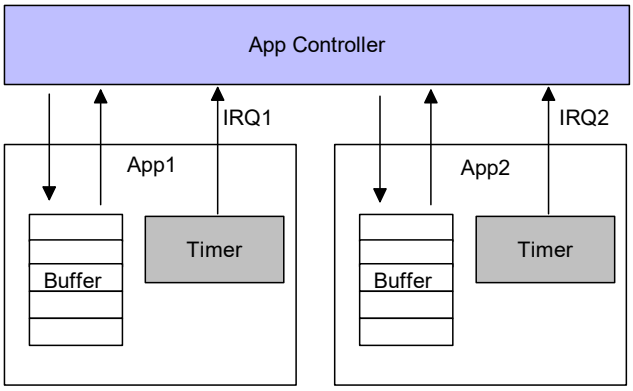


Figure 14 Request turnaround time comparison between the basic design and improved design (see online version for colours)

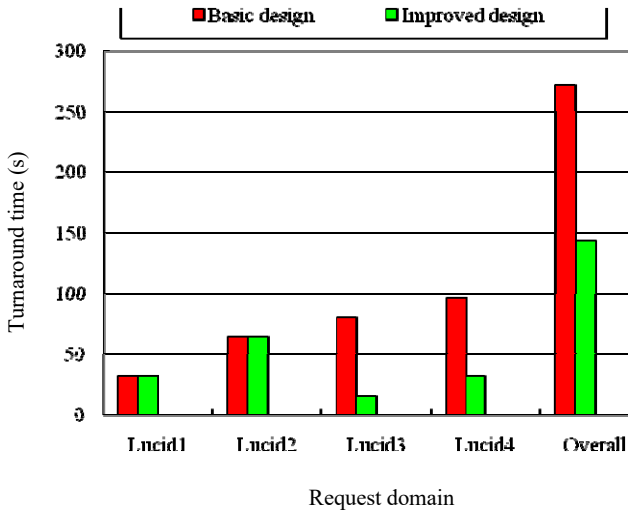


Figure 14 shows the turnaround time when each DomU sends a request with eight blocks of data (32 KB) to the FPGA accelerator simultaneously. The improved design does not affect Lucid2 because its request is inserted into the same queue as the Lucid1 request, and requests in the same queue are scheduled via FCFS policy. The most apparent improvement (an 80% reduction of the basic design's turnaround time) is when the Lucid3 request is inserted into a different queue than the Lucid1 request. Hyper-Requesting helps reduce the scheduling delay caused by scheduling the requests from Lucid1 and Lucid2. The concurrent share of the DMA read channel enables simultaneous use of app1 and app2 on the FPGA accelerator. With the improved design, the turnaround time of all the requests is reduced to approximately 53% of that with the basic design.

5 Related works

5.1 FPGA virtualisation

El-Araby et al. (2008) describe a virtualisation solution for FPGA accelerators. The authors have proposed a solution for virtualising an FPGA accelerator for multiple processes on a single OS. An important component in their virtualisation solution is what they refer to as a 'virtual coprocessor monitor (VCM)', which multiplexes requests from multiple processes to access the shared FPGA accelerator. When a user process sends a request to the VCM to access the FPGA accelerator, the VCM creates a virtual memory space. It shares this space with the calling process through conventional POSIX memory sharing inter-process communication (IPC) primitives. Then the user process copies data to the virtual memory space, followed by the insertion of a request into the request queue. When the FPGA accelerator has finished, the VCM will send an acknowledge signal to the user process indicating that results are ready to be fetched. The VCM also provides an API for a process to release the virtual memory space.

The work presented in El-Araby et al. (2008) does not include a design of an FPGA accelerator, and the FPGA design they use is vendor proprietary. Thus, their proposed VCM, which is designed as a user-level application, interacts with the FPGA accelerator driver through vendor-supplied APIs. The authors modified the vendor-supplied APIs with virtualisation APIs, which can be used by user processes to interact with the VCM. In comparison to the work presented in El-Araby et al. (2008), pvFPGA moves a step forward to virtualise an FPGA accelerator for processes from different domains. We have proposed an accelerator design which can be used for accelerating various applications, regardless of the application computation latencies. Also, our device driver level design solution can be easily optimised (e.g., hyper-requesting).

Byma et al. (2014) focus on integrating FPGAs into OpenStack. Taking advantage of partial reconfiguration, they partition an FPGA into several reconfigurable regions, with each region exposed to OpenStack as an allocable

resource. No novel VMM layer design is presented in this work.

5.2 GPU virtualisation

A direct comparison between pvFPGA and the recent GPU virtualisation solutions may seem unfair, since GPUs have been used for general purpose computing. But pvFPGA draws on some of the techniques used in GPU virtualisation solutions. Our proposed accelerator enables FPGAs to be used for general purpose computing when partial reconfiguration is integrated in the future. GViM (Gupta et al., 2009), vCUDA (Shi et al., 2012) and gVirtuS (Giunta et al., 2010) are three GPU virtualisation solutions that have been recently proposed. In this section, we compare the difference between pvFPGA and the recent GPU virtualisation solutions only in terms of the virtualisation techniques being used.

GViM is a Xen-based system design solution that permits users to run any CUDA-based applications in a domain (Gupta et al., 2009). It uses the shared memory mechanism for data transfer between domains, and an interposer library in each unprivileged domain (DomU) provides CUDA accesses. CUDA calls from user applications in a DomU are intercepted, packed together with parameters into CUDA call packets by the interposer library, and then passed to the frontend driver, which transfers them to the Domain 0 (Dom0) backend driver. In Dom0, the CUDA call packets are continuously passed to the library wrapper, which converts them into standard CUDA function calls. Each DomU has a dedicated CUDA call buffer in Dom0. CUDA calls from DomUs requesting access to the GPU are stored in the buffer first, the buffered requests are then scheduled to be translated into CUDA function calls in a round-robin fashion.

In some respects, vCUDA functions in a similar way as GViM; for example, the latest version of vCUDA also uses shared memory for data transfer between domains. CUDA calls in a DomU are intercepted and packed by a vCUDA library. The vCUDA library has a vGPU component that reveals the device information (e.g., GPU memory usage, texture memory properties) to applications in a DomU. To ensure information consistency, vCUDA includes a synchronisation mechanism between the vGPU and a component in Dom0 known as vCUDA stub. The vCUDA library has a global queue for storing all the packed packets, which are periodically transferred to the vCUDA stub. The vCUDA stub unpacks the received packets and invokes the related CUDA API calls in Dom0. The designers also propose using a Lazy Remote Procedure Call (RPC) mechanism to batch specific RPCs, thereby reducing the number of expensive world switches (context switches between different domains). vCUDA uses working/service threads [introduced in Shi et al. (2012)] to enable requests from the same or different domains to be concurrently executed on the GPU accelerator.

gVirtuS is a VMM independent solution for GPU virtualisation in a cluster environment. Intercepted CUDA calls in a DomU are redirected to the host domain running on a different physical machine via a TCP/IP-based communicator. A resource sharing framework was proposed as an extension of gVirtuS in Ravi et al. (2011). Ravi et al. (2011) created a virtual process context to consolidate different applications (including from different domains) into a single application context, in order to time share or space share streaming multiprocessors (SMs) in a GPU accelerator.

Both the recent GPU virtualisation solutions and pvFPGA use shared memory for inter-domain data transfer, but the above solutions of intercepting user space API calls from frontend domains and redirecting them to the backend domain are specific to GPUs. Owing to the limited knowledge of GPU hardware specifications and the complexity of GPGPU programming model, it is not feasible to achieve GPGPU virtualisation at the low device driver layer, which can provide lower overhead and higher efficiency (i.e., we propose hyper-requesting to reduce request turnaround time). A CUDA application might entail calling thousands of CUDA APIs (Shi et al., 2012), whereas only one ‘call’ is required to access the FPGA accelerator in pvFPGA. The overhead on GPU virtualisation solutions average 11%, whereas pvFPGA overhead is near zero when a DomU uses a large data pool (e.g., 4 MB) for data transfer. Also, no GPU virtualisation solutions include a scheme to supply DomUs with different maximum data transfer bandwidths. The coprovisor efficiently multiplex requests to access the FPGA accelerator at the device driver layer, and pvFPGA can provide different maximum data transfer bandwidths for DomUs by regulating the size of the shared data pools.

6 Conclusions and future works

In this paper, we present an ameliorated design of pvFPGA, which is a leading edge system design solution of virtualising an FPGA-based hardware accelerator by a VMM on the x86 platform. The proposed accelerator design can be used for accelerating various applications, regardless of the application computation latencies on the FPGA. A streaming pipeline technique is adopted in pvFPGA for efficient data transfer between the server and the FPGA accelerator. We discuss the concept of streaming pipeline hazards, and several solutions for solving streaming pipeline hazards are covered in this paper. Additionally, we propose a technique called hyper-requesting, which enables portions of two requests to be simultaneously processed on the FPGA accelerator through DMA context switches, to achieve request level parallelism.

An important future direction of our work is to integrate partial reconfiguration into pvFPGA. With partial reconfiguration, various accelerator applications can be preconfigured as partial bitstream files, and be dynamically swapped into either app1 or app2 in Figure 2(a) according to a DomU’s request, which will accomplish runtime general purpose computing. Another future direction of our work is to extend pvFPGA to a cluster environment, where each node in the cluster is equipped with an FPGA

accelerator. When the coprovisor recognises that there are too many requests in the queue for the native FPGA accelerator, while the FPGA accelerators in other servers are underutilised, some of the requests can be scheduled through an extra TCP/IP mechanism to the idle FPGA accelerators in remote physical machines. This would also raise some load balancing issues that need to be solved.

References

Agrawal, D., Das, S. and Abbadi, A.E. (2011) 'Big data and cloud computing: current state and future opportunities', in *Proceedings of the 14th International Conference on Extending Database Technology, EDBT'11*, pp.530–533.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A. (2003) 'Xen and the art of virtualization', in *Proceedings of the 19th ACM Symposium on Operating Systems Principles, SOSP'03*, pp.164–177.

Brodtkorb, A.R., Hagena, T.R. and Sætra, M.L. (2013) 'Graphics processing unit (GPU) programming strategies and trends in GPU computing', *Journal of Parallel and Distributed Computing*, Vol. 73, No. 1, pp.4–13.

Bugnion, E., Devine, S., Rosenblum, M., Sugerman, J. and Wang, E.Y. (2012) 'Bringing virtualization to the x86 architecture with the original VMware workstation', *ACM Transactions on Computer Systems*, Vol. 30, No. 4, pp.12:1–12:51.

Byma, S., Steffan, J.G., Bannazadeh, H., Garcia, A.L. and Chow, P. (2014) 'FPGAs in the cloud: booting virtualized hardware accelerators with OpenStack', *International Symposium on Field-Programmable Custom Computing Machines, FCCM'14*, May, pp.109–116.

Canny, J. and Zhao, H. (2013) 'Big data analytics with small footprint: squaring the cloud', in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'13*, pp.95–103.

Che, S., Li, J., Lach, J. and Skadron, K. (2008) 'Accelerating compute intensive applications with GPUs and FPGAs', in *Proceedings of the 6th IEEE Symposium on Application Specific Processors, SASP'08*, pp.101–107.

Chisnall, D. (2007) *The Definitive Guide to the Xen Hypervisor*, Prentice Hall, New Jersey, USA.

Chung, E.S., Milder, P.A., Hoe, J.C. and Mai, K. (2010) 'Single-chip heterogeneous computing: does the future include custom logic, FPGAs, and GPGPUs', in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO'10*, pp.225–236.

Cope, B., Cheung, P.Y.K., Luk, W. and Witt, S. (2005) 'Have GPUs made FPGAs redundant in the field of video processing?', in *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology (FPT'05)*, pp.111–118.

Dean, J. and Ghemawat, S. (2008) 'MapReduce: simplified data processing on large clusters', *Communications of the ACM*, Vol. 51, No. 1, pp.107–113.

Dowty, M. and Sugerman, J. (2009) 'GPU virtualization on VMware's hosted I/O architecture', *ACM Operating Systems Review*, Vol. 43, No. 3, pp.73–82.

Eguro, K. and Venkatesan, R. (2012) 'FPGAs for trusted cloud computing', in *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications, FPL'12*, August, pp.63–70.

El-Araby, E., Gonzalez, I. and El-Ghazawi, T. (2008) 'Virtualizing and sharing reconfigurable resources in high-performance reconfigurable computing systems', in *Proceedings of the International Workshop on High Performance Reconfigurable Computing Technology and Applications, HPRCTA'08*, pp.1–8.

Giunta, G., Montella, R., Agrillo, G. and Coviello, G. (2010) 'A GPGPU transparent virtualization component for high performance computing clouds', in *Proceedings of International Euro-Par Conference, EuroPar'10*, pp.379–391.

Gonzalez, I., Lopez-Buedo, S., Sutter, G., Sanchez-Roman, D., Gomez-Arribas, F.J. and Aracil, J. (2012) 'Virtualization of reconfigurable coprocessors in HPRC systems with multicore architecture', *Journal of Systems Architecture*, Vol. 58, Nos. 6–7, pp.247–256.

Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V. and Ranganathan, P. (2009) 'GViM: GPU-accelerated virtual machines', in *Proceedings of ACM Workshop System-Level Virtualization for High Performance Computing, HPCVirt'09*, pp.17–24.

Hadoop [online] http://hadoop.apache.org/ (accessed 12 September 2015).

Huang, C.H. and Hsiung, P.A. (2013) 'Virtualizable hardware/software design infrastructure for dynamically partially reconfigurable systems', *ACM Transactions on Reconfigurable Technology and Systems*, Vol. 6, No. 2, pp.11:1–11:18.

Huang, C.H., Hsiung, P.A. and Shen, J.S. (2010) '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.

Jain, A.K., Pham, K.D., Cui, J., Fahmy, S.A. and Maskell, D.L. (2014) 'Virtualized execution and management of hardware tasks on a hybrid ARM-FPGA platform', *Journal of Signal Processing Systems*, Vol. 77, Nos. 1–2, pp.61–76.

Jan, K., Fan, C., Kuo, A., Yen, W. and Lin, Y. (2015) 'A platform based SOC design methodology and its application in image compression', *International Journal of Embedded Systems*, Vol. 1, Nos. 1–2, pp.23–32.

Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A. (2007) 'KVM: the Linux virtual machine monitor', in *Proceedings of the Linux Symposium*, pp.225–230.

Lagar-Cavilla, H.A., Tolia, N., Satyanarayanan, M. and de Lara, E. (2007) 'VMM-independent graphics acceleration', in *Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE'07*, pp.33–43.

Lee, P.S., Lee, C.S. and Lee, J.H. (2013) 'Development of FPGA-based digital signal processing system for radiation spectroscopy', *Radiation Measurements*, Vol. 48, No. 1, pp.12–17.

Lübbers, E. (2010) *Multithreaded Programming and Execution Models for Reconfigurable Hardware*, PhD dissertation, Computer Science Department, University of Paderborn.

Neiger, G., Santoni, A., Leung, F., Rodgers, D. and Uhlig, R. (2006) 'Intel virtualization technology: hardware support for efficient processor virtualization', *Intel Technology Journal*, Vol. 10, No. 1, pp.3:167–3:178.

Northwest Logic Corporation (2010) *DMA Back-End Core User Guide* [online] https://nwlogic.com/products/docs/DMA_Back-End_Core.pdf (accessed 12 September 2015).

Okuyama, T., Ino, F. and Hagihara, K. (2012) 'A task parallel algorithm for finding all-pairs shortest paths using the GPU', *International Journal of High Performance Computing and Networking*, Vol. 7, No. 2, pp.87–98.

Owens, J.D., Luebke, D. and Govindaraju, N. (2007) 'A survey of general-purpose computation on graphics hardware', *Computer Graphics Forum*, Vol. 26, No. 1, pp.80–113.

Ravi, V.T., Becchi, M., Agrawal, G. and Chakradhar, S. (2011) 'Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework', in *Proceedings of the International Symposium on High Performance Distributed Computing, HPDC'11*, pp.217–228.

Rostrup, S., Srivastava, S. and Singhal, K. (2013) 'Fast and memory-efficient minimum spanning tree on the GPU', *International Journal of Computational Science and Engineering*, Vol. 8, No. 1, pp.21–33.

Sabeghi, M. and Bertels, K. (2009) 'Toward a runtime system for reconfigurable computers: a virtualization approach', in *Design, Automation and Test in Europe, DATE'09*, pp.1576–1579.

Shi, L., Chen, H. and Sun, J. (2012) 'vCUDA: GPU accelerated high performance computing in virtual machines', *IEEE Transactions on Computers*, Vol. 61, No. 6, pp.804–816.

Silva, M.L. and Ferreira, J.C. (2006) 'Support for partial run-time reconfiguration of platform FPGAs', *Journal of Systems Architecture*, Vol. 52, No. 12, pp.709–726.

Taibo, J., Gulias, V.M., Montero, P. and Rivas, S. (2011) 'GPU-based fast motion estimation for on-the-fly encoding of computer-generated video streams', in *Proceedings of the 21st International Workshop on Network and Operating Systems Support for Digital audio and Video, NOSSDAV'11*, pp.75–80.

Tian, X. and Benkrid, K. (2010) 'High-performance quasi-Monte Carlo financial simulation: FPGA vs. GPP vs. GPU', *ACM Transactions on Reconfigurable Technology and Systems*, Vol. 3, No. 4, pp.1–22.

Walters, J.P., Chaudhary, V., Cha, M., Guercio Jr., S. and Gallo, S. (2008) 'A comparison of virtualization technologies for HPC', in *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications, AINA'08*, pp.861–868.

Wang, W., Bolic, M. and Parri, J. (2013) 'pvFPGA: accessing an FPGA-based hardware accelerator in a paravirtualized environment', in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS'13*.

Xilinx Corporation (2010) *Virtex-6 FPGA Integrated Block for PCI Express* [online] http://www.xilinx.com/products/intellectual-property/v6_pci_express_block.html#overview, (accessed 12 September 2015).

Xilinx Corporation (2011a) *LogiCORE IP Fast Fourier Transformv7.1* [online] http://www.xilinx.com/support/documentation/ ip_documentation/xfft_ds260.pdf (accessed 12 September 2015).

Xilinx Corporation (2011b) *LogiCORE IP FIFO Generator v8.2* [online] http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v8_2/fifo_generator_ds317.pdf (accessed 12 September 2015).

Yamaoka, K., Morimoto, T., Adachi, H., Koide, T. and Mattausch, H.J. (2006) 'Image segmentation and pattern matching based FPGA/ASIC implementation architecture of real-time object tracking', in *Proceedings of the 2006 Asia and South Pacific Design Automation Conference, ASP-DAC'06*, pp.176–181.

Notes

1 Streaming pipeline hazards are different from data hazards in an instruction pipeline, since there are no data dependencies between the two blocks of data that cause streaming pipeline hazards.