Enabling FPGAs in the Cloud

Fei Chen[1], Yi Shan[2], Yu Zhang[1], Yu Wang[2], Hubertus Franke[3], Xiaotao Chang[1], Kun Wang[1]

[1]IBM China Research Lab
Beijing, China
{uchen, zhyu, changxt, wangkun}@cn.ibm.com

[2]Tsinghua University
Beijing, China
shany04@gmail.com, yu-wang@mail.tsinghua.edu.cn

[3]Thomas J. Watson Research Center
New York, USA
frankeh@us.ibm.com

ABSTRACT

Cloud computing is becoming a major trend for delivering and accessing infrastructure on demand via the network. Meanwhile, the usage of FPGAs (Field Programmable Gate Arrays) for computation acceleration has made significant inroads into multiple application domains due to their ability to achieve high throughput and predictable latency, while providing programmability, low power consumption and time-to-value. Many types of workloads, e.g. databases, big data analytics, and high performance computing, can be and have been accelerated by FPGAs. As more and more workloads are being deployed in the cloud, it is appropriate to consider how to make FPGAs and their capabilities available in the cloud. However, such integration is non-trivial due to issues related to FPGA resource abstraction and sharing, compatibility with applications and accelerator logics, and security, among others. In this paper, a general framework for integrating FPGAs into the cloud is proposed and a prototype of the framework is implemented based on OpenStack, Linux-KVM and Xilinx FPGAs. The prototype enables isolation between multiple processes in multiple VMs, precise quantitative acceleration resource allocation, and priority-based workload scheduling. Experimental results demonstrate the effectiveness of this prototype, an acceptable overhead, and good scalability when hosting multiple VMs and processes.

Keywords

FPGA virtualization, Cloud, Reconfiguration and Heterogeneous computing.

1. INTRODUCTION

Cloud computing is becoming a major trend for delivering and accessing computation resources and services on demand over the network. Dominant enterprise IT vendors, e.g. IBM, Oracle, Cisco, etc., and Web companies (e.g. Amazon, Google) have all delivered cloud solutions (Pure, ExaData, UCS, AWS, Google Compute Engine) leading the transformation from traditional IT infrastructures to the cloud.

Meanwhile, FPGAs are attractive in many computational domains because of their ability to get performance close to ASIC technology, achieving high throughput and predictable latency, while simultaneously providing better programmability and low power consumption. These features can create significant business value as they enable acceleration technologies that require flexibility and time-to-market which cannot be achieved by a general processor design. More and more business systems are deploying FPGAs as a critical component to achieve overall system performance. Examples are Netezza in enterprise databases [4], Convey in HPC and big data analytics [3], Maxeler in financial data processing [22], and Solace in message middleware [5], to mention a few. Continuous advances in FPGA technology that will further drive their adoption.

Modern FPGA chips provide ever richer programmable logic resources of miscellaneous types, which leads to unprecedented computing capacity of a single FPGA chip. As an example, a Xilinx Virtex-7 2000T (2011) can host 120 AES crypto accelerators or 260 ARM7 processor cores. Partial reconfiguration techniques allow a FPGA chip to be partitioned into regions. Circuits in one region can be reconfigured at runtime without interfering with circuits in other regions, significantly increasing the flexibility of FPGA usage. To increase development productivity and shorten the time-to-market, novel high-level programming models and toolchains have been introduced, including OpenCL [19], Lime [8], etc.

With the simultaneous rise in popularity for both the cloud and FPGAs, it is to be expected that the demand for deploying FPGA-based applications in cloud environments will grow. As a reference, a similar demand for GPGPUs was previously observed and nowadays GPGPU resources are common in cloud environments, e.g. AWS. Our work treats FPGAs as programmable resources that can be reconfigured as on-demand devices. However, integrating such kind of FPGA resources into the cloud is nontrivial. According to our analysis, there are four fundamental requirements that need to be addressed. These are now introduced at a high level and then described further on in more detail.

1. **Abstraction**: FPGAs must be exposed to the cloud stack as a resource pool that can be actively managed, i.e. it can be requested, allocated and deallocated by a tenant. Its usage must be tracked in order to facilitate billing that is associated with the public cloud model. In addition, once

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Computing Frontiers '14 Cagliari, Italy
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

provided to a tenant, the FPGA must be programmable by the tenant similar to other resources such as CPUs and GPUs. However, traditional system software stacks, i.e. operating system and hypervisor, consider FPGAs only as fixed-functional acceleration devices while ignoring their nature of programmability.

2. **Sharing**: Sharing and isolation of resources is a desired and natural requirement for all resources in the cloud. FPGA resources should follow this model and enable sharing among multiple tenants and their applications in order to maximize resource utilization. This obvious trend of sharing is most recently visible in the GPU domain where GPUs were initially limited to only one user/tenant per host. However this restriction has eased as Nvidia now provides hardware support for multi-tenancy in its latest Kepler architecture GPU [21].

3. **Compatibility**: Users have dependencies on the ecosystem (tools, libraries) that support FPGA usage. Typically there is a tight coupling between specific FPGAs, their tool chains and the applications and libraries that are written for a specific FPGA. In many cases there are no standard application binary interfaces (ABI) yet released. To enable transition into the cloud, one must provide the ecosystem and the SDKs in the same manner as they are available in stand alone environments.

4. **Security**: One of the most quoted concerns for a faster cloud adoption is security. Sharing of resources depends on proper isolation. As FPGAs were not designed for multi-tenancy but for single users, security features must be introduced in order to enable FPGA usage in the cloud. FGPA accelerators typically run with full hardware access and hence a single malicious tenant can bring down a complete shared compute host. Though there are techniques to ease the impact and initially enable FPGAs in the kernel, security is best addressed by FPGA manufacturers through additional hardware changes. This paper identifies some of the existing problems and does some initial but limited work to address security with the current available technology.

We believe solving the above four requirements are necessary steps for cloud enablement and we focus on them in this paper. Additional problems, such as FPGA scalability among multi-nodes, are not discussed in this paper. The contributions of this paper can be summarized as follows:

(i) Four major requirements for enabling FPGAs into the cloud are analyzed. This paper not only discusses the problems with current techniques, but also provides guidance for enabling FPGAs in cloud.

(ii) An accelerator pool (AP) abstraction is proposed, which abstracts FPGA as a consumable resource while avoiding hardware dependencies of current FPGA techniques.

(iii) A hardware and software co-design is provided as the framework for integrating FPGAs in a cloud. A service logic (SL) is introduced into a FPGA chip to help address the requirements.

(iv) A prototype of the framework is implemented on an x86-based Linux-KVM environment with attached Xilinx FPGAs, and deployed in a modified OpenStack [2] cloud environment. Experiments are conducted to both verify functional correctness and evaluate performance of the prototype. The overhead introduced by our framework, when compared to native dedicated execution, is less than 10% in most cases and at most 25% for small workload data sizes. Meanwhile the latency overhead introduced by virtualization for each acceleration job is approximately 4 microseconds.

The remaining part of the paper is organized as follows: Section 2 introduces related work. Section 3 provides details and solutions for the four topics mentioned in section 1. Section 4 details the design of our framework for FPGA cloud enablement and describes the prototype implementation. Section 5 discusses experiments and section 6 concludes the paper.

2. RELATED WORK

There are a number of research efforts attempting to make FPGAs general- purpose computation resources. BORPH [9] is a well known project working on providing an OS for creating hardware-based processes and providing FPGA hardware abstractions and management. This work provides interfaces for a hardware thread in the FPGA, so that the hardware thread operates like a software thread on the CPU. Several works about multitasking on FPGA coprocessors or reconfigurable hardware (RH) within processors have been published, such as [27, 13, 25]. These projects focused on accelerator scheduling in reconfigurable hardware based on non-virtualized environments. They provide valuable experiences on resource scheduling, however this is not the fundamental problem when enabling FPGA in cloud. Other previous work refers to terms like FPGA virtualization and virtualized reconfigurable hardware, such as [23], as transparently scheduling and reconfiguring FPGA to multiple jobs as FPGA virtualization. However this is not done in the context of cloud requirements.

Providing an abstraction layer for FPGAs was proposed in both [20] and [16]. Such a layer provides a decoupling between FPGA logics and software, which implies good portability for both FPGA logics and applications. The work in [12] discussed FPGA being used in a distributed system. However neither virtualization nor quantitative resource allocation is addressed. [26] presented FPGAs working in Xen virtualized environment in a single computing node, though not addressing the FPGA dynamical configuration and cloud requirements.

Various papers discussed the security problem with FPGAs in the cloud. For example, the paper [11] proposed a method to configure private accelerators into FPGA if the user did not trust the system administrator. It focused on identifying the accelerators but not on the behavior of accelerators. [6] used FPGAs to build a secure database application, focusing on the security of applications but not the security of the system.

Another approach focuses on using a high-level programming model, e.g. OpenCL and Lime [7], to abstract FPGA resources. Although not directly helping FPGA resources to be shared in the cloud, they can help to grow broader acceptance and utilization for FPGAs and help build a protable ecosystem.

Resource management of GPGPUs attracts significant attention. Both PTask [24] and Gdev [18] are representative works for extending OSs and placing GPGPUs directly under OS management.

Pegasus [15] considers GPGPUs as peers to CPUs and provides a unified resource abstraction and workload scheduling mechanism. However, from an architecture perspective, a GPU consists of 100s of homogeneous cores, which is different from a FPGA as a system consisting of heterogeneous

resources. This difference implies that we cannot simply reuse GPGPU resource management mechanisms for FPGA. To achieve cloud resource abstraction and sharing, critical functions must be implemented as logic in a FPGA chip, which is not possible for commodity GPGPUs.

3. ENABLING FPGA IN CLOUD

Four key requirements have been raised in Section 1 for FPGA enablement in a cloud and they are analyzed and addressed in this section in the context of our framework, as shown in Figure 1.

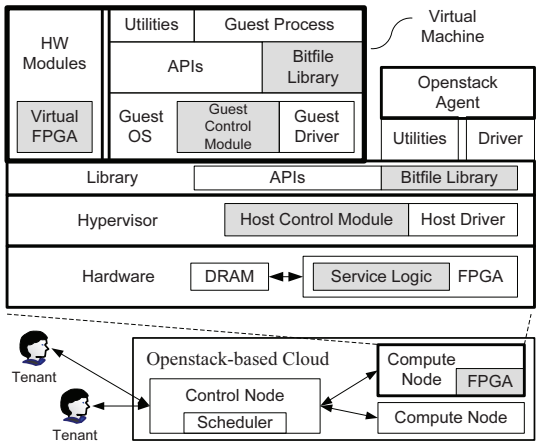

Figure 1: FPGA framework in a cloud

Except for the modules in gray, Figure 1 shows the typical components of an OpenStack-based cloud. A number of compute nodes, each of which is a physical machine, provide physical resources including CPUs, memory, disks and networking. A control node handles requests from tenants, schedules resources and creates virtual machines (VMs) on selected physical machines. Tenants then get access to their virtual machines and deploy applications on them.

In order to introduce FPGAs into a cloud, we provide new modules to the compute nodes, as shown in grey in Figure 1. In addition, the scheduler in the OpenStack control node is enhanced. These modules will be described in this section, and the implementation details will be introduced in Section 4. To facilitate our description, we divide the system stack of the compute nodes into 4 layers: hardware, hypervisor, library and application.

3.1 FPGA Resources Abstraction

In order for FPGAs to become resources that can be consumed by tenants, miscellaneous types of resources in FPGA chips and FPGA-centric cards must be abstracted into a manageable resource pool, similar to CPU and memory resources. A straight-forward approach for abstracting FPGAs is a programmable resource pool (PRP), in which FPGAs are abstracted into an set of resources including registers, LUTs, memory, etc. and placed under the management of the cloud system. A tenant can issue requests for an arbitrary amount of registers, LUTs, and memory and the the controller provides a virtual FPGA chip consisting of what was requested. The tenant can work with the virtual FPGA , e.g. developing logics following a normal design flow, unaware of the virtualization taking place in the cloud environment.

Although the PRP abstraction aligns well with the nature of both FPGA and cloud, it is not feasible in current clouds. The reason relates to the current FPGA chips capabilities and their development tool chains. To generate a FPGA bitfile, real hardware details of an FPGA chip, including its type and specification, which area in the chip the bitfile will be configured into, etc. must be exposed to the current tool chain. Also, a generated bitfile can only be configured into a specific area within a specified type of the FPGA chip. Those restrictions introduce security concerns and flexibility problem in the cloud. Therefore, unless FPGA chips and tool chains support generating hardware-independent bitfiles, i.e. support stronger virtualization, the PRP abstraction is hard to consume in the cloud.

As an alternative, we propose an accelerator pool (AP) abstraction as a trade-off between current FPGA limitations and cloud principles. In the AP abstraction, each FPGA chip has several pre-defined accelerator slots, e.g. slots A, B, C and D shown in Figure 2. By using the dynamic partial reconfiguration mechanism of modern FPGAs, each slot can be considered as a virtual FPGA chip with standardized resource types, capacity and interfaces. Therefore, each slot can only host an accelerator with compatible resource requirements and interface design. In non-cloud environments, [10] and [14] discuss usage of this feature. Using AP, FPGA chips become a pool of accelerators with various functions and performance. Instead of requesting programmable resources in PRP, a tenant directly requests various combination of accelerator functions and performance. A cloud provides a list of pre-defined accelerators, handles tenant requests and configures accelerators into idle slots. If no accelerator matches the requirements, a tenant can submit his own designs and the cloud owner performs the compilation and adds the tenant design into the accelerator list.

Due to a level of *standardization*, AP provides a more feasible and consumable approach to introducing FPGAs into the cloud. An accelerator design following standards can be mapped into standard slots. The designer cares about the specification of slots, instead of FPGA chip details. The cloud owner will compile various designs for all compatible slots with a predictable cost. Also, such mechanism enables an application utilizing FPGAs to be deployed without location dependency. Additionally, cloud systems can approach resource management more easily because standard slots lead to regular resource allocation. Additional advantages of AP is that both accelerator bandwidth and acceleration job priorities become manageable resources. A major limitation of AP is that tenants can not generate bitfiles and configure them into FPGAs as the FPGA hardware details are transparent. This limitation can not be removed before tool chains generate hardware independent bitfile.

To support AP, a service logic (SL) module inside the FPGA is introduced in the hardware layer of our system stack. The SL provides standard interfaces for in-slot accelerators. It facilitates accelerator bandwidth and priority management. In the cloud, the control node selects appropriate compute nodes that have the desired available FPGA resources for tenants, and the host control module(HCM) in the selected node finds FPGA slots and configures accelerators via the service layer. Inside VMs, a guest control module(GCM) is introduced to achieve accelerator bandwidth and priority management.

3.2 FPGA Sharing

Given the above description of FPGA slots and assignment to different tentants, sharing of FPGA resources is a key requirement. FPGA sharing by multiple applications has been discussed in [10, 28, 17]. With sharing physical resources security issues arise. Proper virtualization support that guarantees isolation among tenants is required. Since FPGAs were designed for single-user cases, no virtualization support exists. Here we propose an accelerator virtualization mechanism.

The main requirement for isolation/virtualization is a secure low overhead address translation between a VM and a host compute node. The issues arise in that VMs use address virtualization, i.e. they are only aware of guest physical addressing (GPA) and are unaware of the host physical addressing (HPA) in the compute node. All buffer and signal addresses in a VM are expressed in GPA, yet the FPGA only utilizes HPAs at this point. Hence a GPA-HPA translation must be introduced to enable accelerators to access VM memory areas in order to fetch and push required data. In our mechanism, we examine two methods for translating addresses and moving data.

The first method copies data between VM memory and host buffers on the compute node. Accelerators only access host buffers for which physical addresses are known. This method, named as *VM-copy*, is easy to implement while incurring data copy cost. Another method is maintaining a fixed mapping relationship between GHA to HPA by the hypervisor. When a VM triggers an accelerator, it will trap into the hypervisor, and GPA is translated into HPA and then sent to an accelerator. This method, named *VM-nocopy*, introduces no data copy overhead. However it requires modifications to the host OS to reserve large blocks of physical memory and modifications to the memory allocation method for VM creation. Both methods are evaluated in this paper.

To implement our virtualization mechanism, a virtual FPGA device model is added to each VM. The model transfers commands, signals and data between the VM and host when necessary. The HCM in the hypervisor layer is enhanced for address translation. Besides, the HCM collaborates with FPGA device model to achieve accelerator utilization tracking for cloud management.

3.3 Compatibility in FPGA

In order to allow accelerators developed by different FPGA developers to exist in the same FPGA platform in the cloud, and in order to decouple accelerator developers and software developers, there should be (i) unified RTL-level interfaces defined to connect an accelerator to the SL in the FPGA, and (ii) unified software-hardware interfaces for software, including both system software and applications, to talk to accelerators through the SL.

Current systems have two methods to define unified software-hardware interfaces, CPU push/poll and DMA. Different kinds of accelerators use different interfaces. For example, an accelerator for graph search will likely use push/poll, while a compression engine will likely take advantage of DMA. In this paper we define the unified software-hardware interfaces based on DMA. APIs are introduced in the library layer and the VMs.

3.4 Security

Some previous works [11, 6] discussed the security problem from different points of view. To enable FPGAs in cloud, there are two fundamental problems to be addressed, preventing users' accelerators crashing the system and preventing the accelerators from stealing and polluting in memory data. The first problem has been discussed in Section 3.1 and 3.3. The second problems can be addressed by a hypervisor and an IOMMU. We assume that the hypervisor and the kernel code in the host machine are trustable, and that the FPGA platform designed by the cloud provider is also trustable.

While in a multi-tenant environment, memory accesses from a CPU are already secure with today's hypervisor enabled architectures, a problem exists in illegal memory accesses from accelerators, as stipulated under the sharing requirement. As tenants can submit their own FPGA accelerators, it must be guaranteed that those accelerators can only access memory associated with the virtual machine owning the accelerator.

IOMMUs, existent in many chip-sets, filter memory accesses from IO devices and provide protection. Only the address space registered by the hypervisor is allowed to be accessed. However, IOMMUs match a memory access by bus number, device number and function number. If an accelerator is shared by two processes, there is still the potential for data leakage and cross partition writes. Therefore we choose DMA as a safer method for data transferring. In our framework, a DMA engine is provided by SL, under control of the trustable hypervisor. All DMA operations are supervised by the hypervisor for parameter checking. Only correct DMA operations are issued. Other undesirable accelerator behaviors include excessive DMA commands to the SL, overusing FPGA internal bandwidth. The SL can apply accelerator bandwidth control to control these situations.

4. PROTOTYPE IMPLEMENTATION

In this section, a prototype implementation of our framework in Section 3 is introduced and technical details are discussed. The prototype is based on X86 physical machines running Linux and KVM-Qemu [1]. An OpenStack is deployed to manage machines with FPGAs as compute nodes. Although the prototype was conducted using a PCIe FPGA card, this paper does not assume that the FPGA is hosted on an IO attached card outside the CPU.

As shown in Figure 1, a compute node in our prototype is comprised of four logical layers, i.e. hardware, hypervisor, library and application. At the hardware layer a FPGA subsystem (PCIe FPGA card) is integrated. The SL provides common and necessary in-hardware support for high-level management capabilities. The hypervisor layer runs directly on the hardware and provides the AP abstraction and virtualization. A library layer is used to manage FPGA bitfiles and provides APIs for the application layer. Various applications interact with the FPGA via interfaces. Here applications mainly include management utilities on compute nodes and tenant's processes in their VMs.

4.1 Hardware Layer

At the hardware layer, FPGAs are physically connected to the existing cloud hardware. This layer defines interfaces to access FPGAs for the three software layers in the framework. Moreover, this layer provides common and necessary mechanisms enabling high-level management operations.

The prototype utilizes a Xilinx FPGA PCIe card. The

SL is implemented in the FPGA chip and a host driver provides a transparent wrapper to both the software stack and hardware logics in FPGAs.

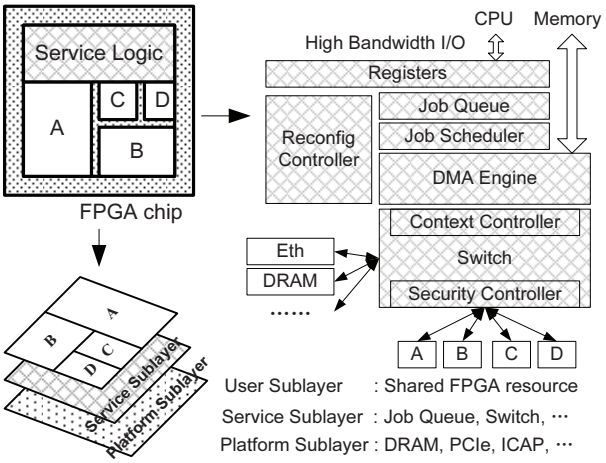

Figure 2: Design in FPGAs

Figure 2 shows the internal logical structures of the FPGA chip in our framework. The FPGA itself can be considered as a subsystem stack, comprised of three sublayers, i.e. the platform sublayer (at the bottom), the service sublayer and the user sublayer (on the top). The platform sublayer is implemented by on-chip fixed-functional circuits supporting accesses to hardware resources attached to the FPGAs, such as memory and Ethernet controllers. Both the service sublayer and the user sublayer reside in the programmable resource area of FPGAs. Partial reconfiguration helps to partition them from each other, so that a logic configuration operation happening at the user sublayer will not interrupt the execution of the service sublayer.

The service sublayer is just the SL in FPGAs, providing both the interfaces for software layers in our framework and mechanisms enabling management operations. SL is the key hardware module in our framework. Details will be discussed below.

The user sublayer provides the resources for cloud tenants using our AP abstraction. The programmable area dedicated to this sublayer is partitioned into a number of standard accelerator slots, e.g. A, B, C and D in Figure 2. Because of current FPGA hardware and toolchain limitation, the slot layout can not be changed unless the whole FPGA chip is reconfigured.

Among those three, the service sublayer is most important. It implements functions to facilitates FPGA subsystem management, and provides all interfaces to address the compatibility problem in Section 3.3. The service sublayer defines ABIs for the software to access FPGAs, including both accelerators and management functions of SL. Also, the service sublayer defines accelerator interfaces so that an accelerator in a slot can be connected with the SL.

Figure 2 shows the basic components comprising the SL. Registers are exposed to the software stack through memory mapped IO (MMIO). A job queue receives acceleration jobs from software. A scheduler manages the job queue and schedules jobs into accelerators according to software-specified strategies, such as priority and workload size. A job context is saved by the job scheduler if the scheduler terminates the job running on a preemptable accelerator. A shared DMA engine is used to fetch and store data to/from accelerators. A switch dispatches and gathers data to/from accelerators for the DMA engine. In order to share the DMA engine for multiple accelerators, multiple DMA contexts, including the buffer's address, size and etc., are maintained in the context controller in the switch module. The kernel logic in a SL instance with two accelerator slot ports are detailed in Figure 3.

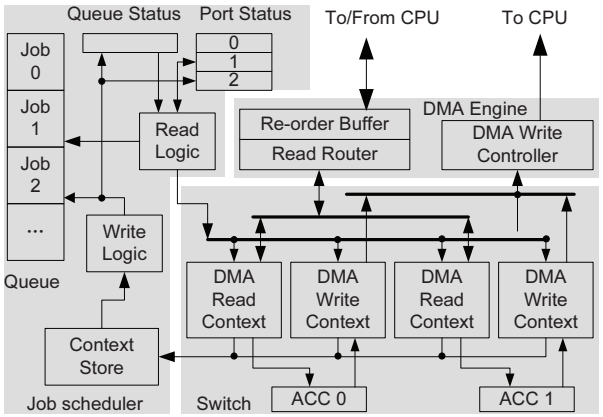

Figure 3: Kernel logic in SL

Besides the kernel logic, a security mechanism is provided in the security controller for each interface between accelerators and switch module. It isolates an accelerator when no job is scheduled on it, detects errors such as time out error and data format error, and finally reports the error to software. The reconfiguration controller (Reconfig controller in Figure 2) receives the command and incremental bitfile from the HCM, and then reconfigures the FPGA through the partial configuration interface in the platform sublayer.

4.2 Hypervisor layer

The FPGA framework provides two modules in the hypervisor layer. One is the host driver providing accessability to the FPGA subsystem. The other module is the HCM which manages FPGA resources and provides the bottom-level software interfaces. This module maintains three data structures for abstraction and sharing purposes. The *FPGA_info* structure records the FPGA chip's static and runtime information, such as the chip specification, the layout and status of in-chip accelerator slots. The *ACC_info* list records information for each accelerator configured in a slot, including a unique accelerator identifier (ACC_ID) and the peak bandwidth. The *Job_info* list records information of all jobs running on each accelerator. By maintaining such data structures, the HCM provides three basic services enabling abstraction and sharing:

For abstraction, the HCM tracks FPGA usage and accelerator status. It records what type of accelerators have been downloaded to accelerator slots, maintains unique ACC_IDs for them, and records the in-flight jobs on the FPGA. When receiving a request for a given type of accelerator, it can find an in-slot accelerator, choose an idle slot to configure a new accelerator or even reject the request when illegal.

For sharing, the HCM supports memory management, including buffer allocations and address mappings for applications. Only the HCM knows about the HPA that the DMA engines need. Buffers needed by applications are allocated here and mapped to the user space for direct application ac-

cess. It also authorizes FPGA management utilities on the application layer to access registers in the FPGA.

Moreover, the HCM provides runtime control and monitoring for sharing. When a VM starts a job, it will trap into the HCM using the *ioctl* call or raise a VM exception, so that the HCM can take the job request and translates DMA addresses for proper hardware access. The HCM monitors the runtime status by accessing performance counters in the FPGA or its local data structures.

The HCM provides a set of commands via *ioctl* calls and shared memory pages for the communication between the HCM and the upper layers in the framework (Figure 1). The HCM will only be invoked at the time of management and job initialization, and as a result it introduces only an overhead of less than 1 microsecond for the job execution.

4.3 Library Layer

The library layer wraps up services from the HCM, provides APIs for applications using accelerators, maintains a bitfile library, and also exposes Linux *ioctl* commands for FPGA management. In our prototype, there are four APIs for programmers, *acc_open*, *acc_do_job*, *acc_wait* and *acc_close*. The *acc_open* opens an accelerator with a given name, and allocates buffers for job parameters, source data and result data, and finally returns a handler containing necessary information. After the source data is available in the buffer, the *acc_do_job* is called to start a job in the FPGA. The FPGA framework moves parameters and source data to the accelerators and the embedded protocol is interpreted by accelerators themselves. The *acc_wait* is used to wait for the completion of the job, and the returned data will provide the job result status. The *acc_close* will free all buffers and delete associated records in the HCM. These APIs free the software and accelerator developers from having to negotiate with each other regarding any protocol, which we consider the basis for a viable FPGA ecosystem.

The bitfile library maintains bitfiles for all supported accelerators. The library includes not only the bitfiles generated by the FPGA development toolchain, but also descriptions for each bitfile. For example, the description includes a unique name/identifier for each accelerator, which slot the accelerator bitfile fits in and the max bandwidth this accelerator provides. The bitfile library is implemented by leveraging the image management mechanism in OpenStack. For VM images, OpenStack has existing mechanisms for their storage, query, authorization and movement to the compute hosts. Only small modifications were needed to enhance OpenStack to support FPGA bitfiles.

When a VM with an accelerator requirement is created, the bitfiles are transferred to the selected compute node together with the image file. An xml file is generated by OpenStack for Qemu to launch the VM. A modified Qemu interprets the xml file with FPGA accelerators descriptions and creates the VM.

4.4 Application Layer

Utilities and the IaaS (Infrastructure as a Service) driver for OpenStack directly run in the application layer. In the OpenStack control node, Horizon (OpenStack web GUI) and the Glance (OpenStack storage component) have been modified to allow administrators the uploading of FPGA bitfiles and the allocation of accelerator quotas for VMs before the VMs start running. The Nova scheduler which is an OpenStack compute component is modified to assign a VM requiring FPGA accelerator to a compute node with FPGA.

As shown in Figure 1, in an OpenStack compute node with FPGA, before the VMs start running, a modified Nova compute component interprets commands from the control node and uses the low-level commands in the library layer to talk to the HCM to create an accelerator, monitoring the status and so on. Utilities have been supported in the prototype. For example, *fpga_addacc* can be used to add an accelerator for the VM, *fpga_bwctl* can be used to set the bandwidth allocation for an application, *fpga_priority* can be used to set the scheduling priority of an application.

For a VM, a GCM (shown in Figure 1) handles requirements and provides a shared and managable FPGA view inside the VM by cooperating with the HCM and a virtual FPGA model(VFM). The GCM provides the same APIs for the guest software stack as what HCM provides to the host software stack. But the implementation of GCM is different from the HCM. Instead of talking to the real FPGA hardware, the GCM accesses the VFM, which is an I/O device simulator added to Qemu. The VFM traps I/O accesses from a VM and maintains the connection between a VM and the host, such as a map between the accelerator ID in the VM and the accelerator ID in the host.

Next we will discuss how an application utilizes an accelerator. For simplicity, we at first describe how an application in a host accesses an accelerator. Then we expand the discussion to the case running in a VM.

Each accelerator job is represented by two buffers and one descriptor. One buffer keeps the source data to be processed by the accelerator, and the other buffer will be filled by the accelerator with the result data. The descriptor is a structure of 64-byte length, and the members of the structure include the job priority, the ACC_ID, the addresses and size of both input data and result buffers, parameters for this job, the address of this descriptor, and the job status.

For a job to be invoked, all members except the data size, parameters and status, are filled by the HCM and protected by a checksum in the HCM, so that applications can not modify members after job submission. An application fills the data_size and parameters before sending the job to the FPGA. The status is initialized by the application and refilled by the FPGA. The status provides information about whether the job has completed normally, whether the result buffer is big enough, what the result size in the result buffer is, and so on.

Figure 4 shows the main software execution flow in the application and HCM. The application talks to the HCM through Linux *ioctl* commands in the library layer. The buffers and descriptors are allocated by the HCM running in kernel mode and then mapped to user space, so that both HCM and application can access the buffers and descriptor. A *Key* is used in the HCM to index the buffers and checksum mentioned above. It is also returned to the application to be used as identification of the job.

The execution flow is almost the same for an application in a VM as it is in a host, except that the GCM talks with both the VFM and the HCM to complete a job.

The GCM asks for the HCM to get information and permission to access accelerators in a physical FPGA. As mentioned in Section 3.2, there are two methods to move data between a VM and a physical FPGA. If VM-copy is used, the VFM will work as an agent copying data between the

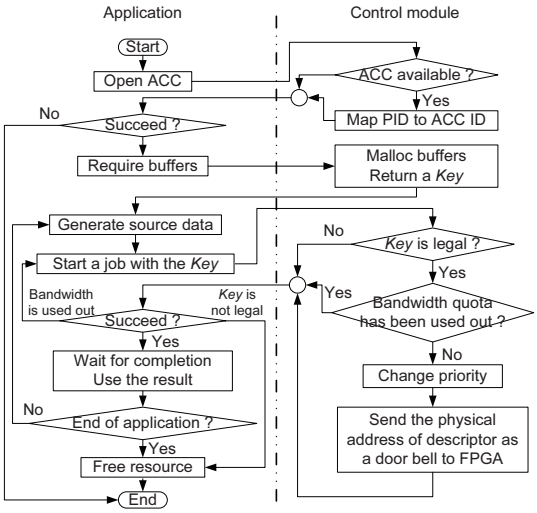

Figure 4: Software execution flow

VM and buffers in the host. For VM-nocopy, the HCM translates GPAs in a VM to HPAs so that the DMA engine can directly access buffers and descriptor in VM memory.

5. EVALUATION

5.1 Evaluation Setup

The evaluation is conducted on an IBM X3650M server with one Intel Xeon X5690 processor. The processor provides 6 cores, 12 hyper threads, and works at a frequency of 3.47GHz. 16GB DDR3 memory is installed in the server and memory operates at 1067MHz. The OS on the server is a Fedora Core 12 with Linux kernel 2.6.31.5. The Qemu used for the VM is a modified qemu-kvm-1.0.

A PCIe card with one Xilinx Kintex-7 XC7K325T FPGA is used as our FPGA subsystem, and the PCIe interface is configured as Gen2, 8 lanes, providing a bandwidth of 5 GB/s, including 8b/10b coding overhead and PCIe transaction overhead. Our system operates at 100MHz. In an instance with an 8-slot job queue and a 3-port switch, the resource usage of our SL is 3,289 Slices and 1,260Kb BRAM, which are only 6.46% and 3.62% of our FPGA chip. Adding one more switch port will use another 290 Slices and 8Kb BRAM.

The proposed FPGA platform treats different types of accelerators as FPGA components requiring different FPGA reconfigurable resources, different I/O bandwidth and different workload patterns. The FPGA platform is not concerned with what accelerators are accessed through the DMA engine. We utilize four accelerators which cover the characteristics to evaluate the proposed FPGA platform. Accelerators for AES, SHA-256, Stereo Matching (Stereo), and Matrix-Vector Multiply (MVM) are selected and their characteristics are listed in Table 1. AES is from the encryption domain and MVM is from the science compute domain. Both are data-intensive, using as much bandwidth as PCIe or the DMA engine can provide. Stereo is a representative case for computation-intensive tests from the computer vision domain, requiring many reconfigurable resources but little I/O bandwidth. SHA is a widely used hashing mechanism. Its FPGA implementation is not pipe-lined, thus requiring limited yet bursty bandwidth to fill its buffer. Among these

Table 1: Accelerator Characteristics

Acc.	BW (GB/s)	Logic (K Slices)	Memory (Mb)	Bitfile (MB)
AES	1.60	3.22	0.13	4.64
SHA	0.09	0.59	0.07	4.64
Stereo	0.20	18.01	2.67	4.64
MVM	1.60	1.31	0.14	4.64
FPGA	1.28	50.95	34.80	11.18

four, only the AES accelerator supports preemption, permitting a job to be interrupted. All accelerators share one DMA engine, of which the peak bandwidth is 1.28GB/s and bandwidth is the only runtime resource shared by multiple accelerators after the FPGA configured.

A micro benchmark using these accelerators is constructed. The benchmark generates random data and issues jobs to accelerators following the execution flow shown in Figure 4. The benchmark can send jobs to accelerators with different payload size, priority and random patterns. To assess the FPGA performance in both physical and virtual environments, the micro benchmark is run in three different environments. The first one is in the host OS. The second one is the VM using VM-copy. The third one is the VM using VM-nocopy. Performance comparison between FPGA accelerators and software implementations is not conducted since that is not the focus of this paper.

Three attributes are used to evaluate the prototype. The first one is bandwidth, which shows how much source data can be computed by an accelerator per second. The second one is latency, which shows how much time will be used to finish a job. The starting point of the measurement is after the source data has been ready in the source buffer, and the end point of the measurement is when the application is notified by the FPGA that the result buffer has been filled with data. The time measurement is done through reading the user-level accessible clock cycle counter inside the CPU core with overhead in the 15 nsec range. The third one is the coefficient of variance (CV) for the job latency in a one second window. CV is the standard deviation of latency divided by the average latency, which describes the jitter while executing many jobs.

5.2 Payloads Evaluation

The AES accelerator is used to provide basic performance numbers, because it uses maximum bandwidth and exhibits the lowest latency for a job. The micro benchmark sends jobs with different payload sizes to the AES accelerator. The performance results are shown in Figure 5. When the payload is small, neither of the test cases can fully utilize the accelerator due to the software overhead being the dominant component. Software overhead in VM-nocopy is larger than that in the host although no memory copy is needed. With 4KB payload, a host application spends 11 us for a round trip including 1.2 us for entering the host control module. A job issued in VM-nocopy forces multiple context switches (application in VM → KVM in host (potentially several times) → guest control module → KVM in host → Qemu in host → host control module). Therefore, about 4 us overhead (from application running in a VM to Qemu) is added as compared to the host application. At payloads larger than 256KB, performance of VMnocopy is virtually the same as the host. For VM-copy, the benchmark will never reach the

peak performance as memory copy overhead also increases with payload increase.

Jitter is typically high in VM environments when the payload is less than 64KB because the request latency is not high enough to ignore the interference from OS scheduling for VMs. The 4KB payload case in VM-nocopy has higher jitter than the case in VM-copy. This is because the jobs in VM-nocopy have low latency hence they are sensitive to interference. The 8KB payload case of VM-copy has the highest jitter. This is due to time spent in the Qemu memory copy function.

The performance numbers suggest that a system with FPGA accelerators might consider a hybrid approach where requests with small latencies are performed by software.

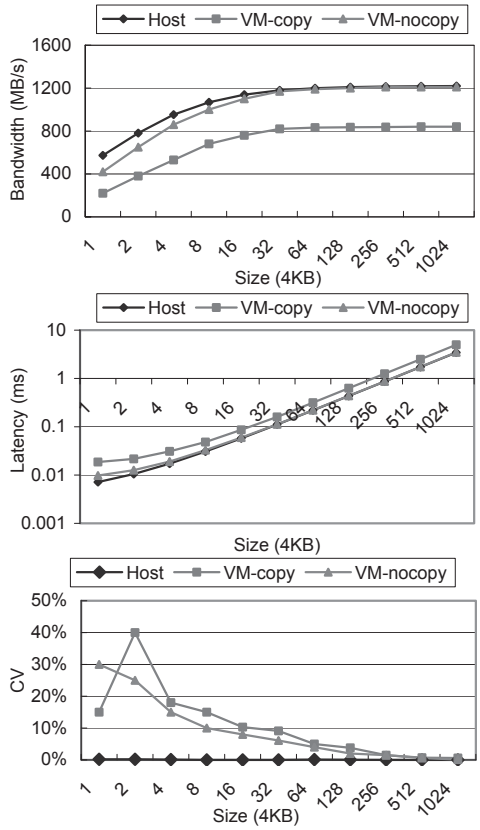

Figure 5: Test with different payload

5.3 Accelerator Sharing

In order to test intra-node scalability and accelerator sharing efficiency, as much as eight processes sharing one and multiple AES accelerators is tested, and the total bandwidth and average latency for those processes are shown in Figure 6. Four scenarios are compared. The first scenario is that all processes run on the host directly and share one AES accelerator ("host"). The second scenario is that all processes run in one VM-nocopy VM and share one AES accelerator ("One VM"). The third scenario is that each process runs in one VM-nocopy VM and multiple such VMs share one accelerator ("VMs"). The final scenario is that each process runs in one VM-nocopy VM and each such VM uses one independent AES accelerator ("AESs"). Each process has a payload of 256KB and the bandwidth and latency results in Figure 6 show that in all environments, the intra-node scalability is good as the overall bandwidth reaches the peak bandwidth that PCIe can provide. Job latency jitter tends to increase when one accelerator is shared by an increasing number of processes. When eight processes use AES with the payload of 256KB in each job, the CV is about 30%.

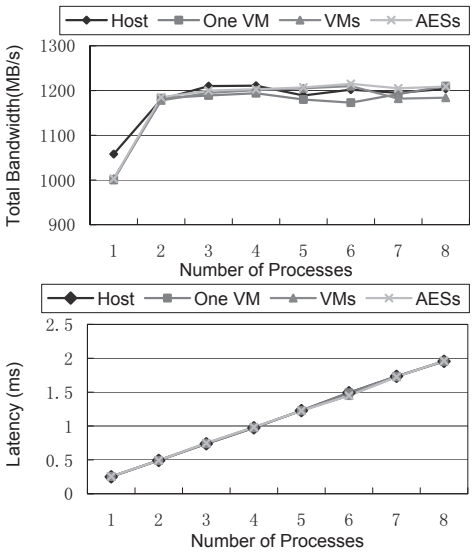

Figure 6: Processes share an accelerator

5.4 Accelerator Management

Bandwidth and priority control are used to show the resource management in our prototype. Figure 7 demonstrates that the bandwidth of two processes using an accelerator can be managed. The micro benchmark using the MVM accelerator runs twice in VM-nocopy. The two processes use MVM with best effort and the source data size is 256KB. The bandwidth for each of the two processes has been tested, and the total bandwidth is also shown in Figure 7. Between second 1 to second 19 , no bandwidth control exists for the VMs, so that each of the processes has a performance of about 608 MB/s, and they are fully using the MVM physical bandwidth. At 19 seconds the *fpga-bwctl* utility metioned in Section 4.4 is used in the host machine to control the bandwidth quota for VMs to 500 MB/s. The two processes in the VM still use AES fairly between second 19 to second 33. At second 33 and second 48, *fpga-bwctl* is used to control the bandwidth quota for process 0 to 100 MB/s and 200 MB/s respectively. This experiment demonstrates that bandwidth is a manageable resource.

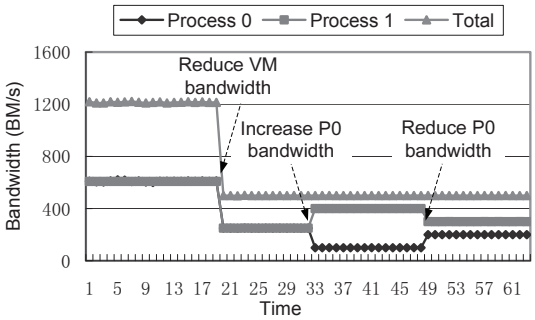

Figure 7: Bandwidth control in prototype

Figure 8 shows that the priority of two processes using an accelerator can be managed. The micro benchmark runs twice in VM-nocopy and sends jobs to one instance of a

preemptable AES accelerator. The process 0 sends 100 jobs to AES randomly in a second, and the data size in a job is 256KB. Process 1 uses AES with best effort, and the source data size is 4MB. Process 1 runs 16 seconds after process 0. Before process 1 starts, the bandwidth of process 0 is about 25 MB/s because it limits its jobs to 100 times a second. The latency and jitter are low in the period as it uses AES exclusively. CV at 9 seconds and 13 seconds are high because applications in the VMs run in the software layer so they are easy to be disturbed. At 16 seconds, process 1 which issues jobs with large payload joins. Process 0 is heavily disturbed because it issues jobs that are much smaller than process 1. The latency of process 0 increases because process 0 and process 1 run with the same priority and the jobs from process 0 have to wait for the completion of the jobs from process 1. At 38 seconds, the *fpga-priority* command is used to raise the priority of process 0. With higher priority, jobs from process 0 can force a context switch in the FPGA so that they can be finished as soon as possible. The results in Figure 8 show that context switching is efficient for the FPGA subsystem in this scenario and priority is also a manageable resource in the FPGA framework.

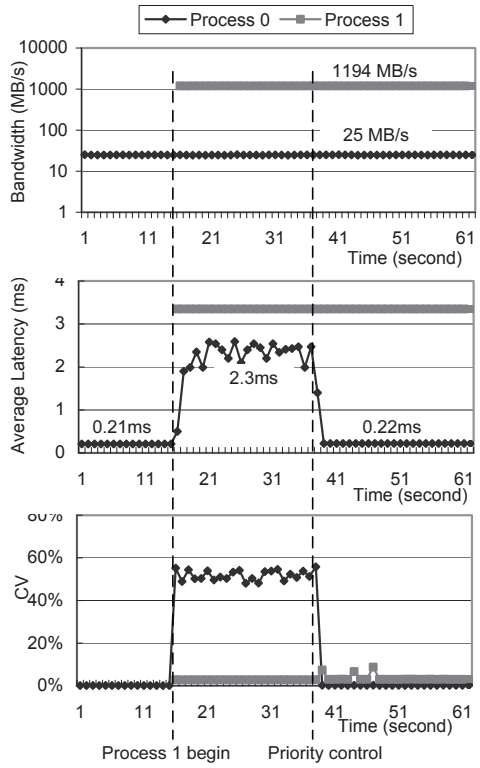

Figure 8: Priority control in prototype

5.5 Accelerator Scheduling

In this section, we show the possibility of increasing overall throughput by scheduling accelerators to efficiently share resources.

Sequential execution is always a baseline for scheduling. Moreover, it is worth noting that data-intensive accelerators might occupy the entire PCIe bandwidth (BW). In such a case, configuring another data-intensive accelerator into another slot does not improve throughput because of the BW bottleneck. In contrast, adding a computation-intensive one might improve overall throughput by doing computation while BW is not available.

Let's suppose four accelerator jobs exist, each for one accelerator type in Table 1 respectively. The payload size of each job is shown in Figure 9.

Three scenarios are used to show the effectiveness of scheduling: (1) Only one accelerator is configured into FPGA at a time and all jobs processed sequentially; (2) Two slots host two accelerators concurrently, while accelerators are scheduled ignoring their data-intensive or computation-intensive nature; (3) Two slots host two accelerators concurrently, and accelerators are scheduled considering their various requirements for computation and I/O resources.

Figure 9 shows the time consuming for each job in three scenarios. The time consuming for each job varies because of the BW conflict. According to the figure, the result in scenario 3 improves 39% and 16% of throughput than that in scenario 1 and 2, respectively, by mixing computation-intensive and data-intensive accelerators. Considering multiple heterogeneous nodes in a cloud, how to schedule different types of accelerators to improve efficiency will be challenging and this will be our future work.

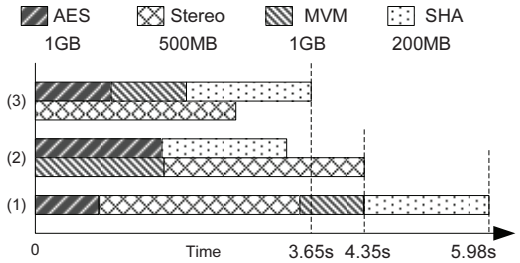

Figure 9: Accelerator scheduling

The evaluations in this section show our basic concept on FPGA in cloud, and also demonstrate the feasibility of the proposed framework, especially for the I/O programming model which is the defacto model in practice today.

6. CONCLUSION

In this paper we analyze the impediments to bringing FPGAs as a shareable resource to the cloud. We further introduce where FPGA manufacturers can provide architectural support to overcome these impediments. We provide a framework and a prototype that provides an FPGA cloud solution in the confines of today's FPGA technology. We propose an AP abstraction for abstracting FPGA resources in the cloud, and introduce an SL as a key hardware module to enable FPGA management in the cloud system stack. Given the prototype, we also demonstrate how abstraction, sharing, compatibility and security can be achieved while using FPGAs in the cloud. Our future work will focus on resource scheduling for large scale heterogeneous cloud computing.

7. ACKNOWLEDGMENTS

The work is supported by IBM, National Natural Science Foundation of China (No.61373026), and Tsinghua University Initiative Scientific Research Program.

8. REFERENCES

[1] Kernel Based Virtual Machine . Website. http://www.linux-kvm.org/page/Main_Page.

[2] Openstack - Open source software for building private and public clouds. Website. http://www.openstack.org/.

[3] Hybrid-Core: The "Big Data" Architecture. Website, March 2013. http://www.conveycomputer.com/files/7013/5075/9401/Hybridcore-The-Big-Data-Computing-Architecture.pdf.

[4] IBM PureData System for Analytics Powered by Netezza technology. Website, 2013. http://public.dhe.ibm.com/common/ssi/ecm/en/imd14400usen/IMD14400USEN.PDF.

[5] Technical publication. Website, March 2013. http://solacesystems.com/library/3200-series.php.

[6] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. A secure coprocessor for database applications. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, Sept 2013.

[7] J. Auerbach, D. Bacon, P. Cheng, R. Rabbah, and S. Shukla. Virtualization of heterogeneous machines. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 890–894, 2011.

[8] Auerbach, Joshua and Bacon, David F. and Cheng, Perry and Rabbah, Rodric. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 89–108, New York, NY, USA, 2010. ACM.

[9] R. Brodersen, A. Tkachenko, and H. Kwok-Hay So. A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph. In *Hardware/Software Codesign and System Synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th International Conference*, pages 259–264, 2006.

[10] L. Chen, T. Marconi, and T. Mitra. Online scheduling for multi-core shared reconfigurable fabric. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 582–585, 2012.

[11] K. Eguro and R. Venkatesan. Fpgas for trusted cloud computing. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 63–70, Aug 2012.

[12] B. S. Frank Opitz, Edris Sahak. Accelerating distributed computing with fpgas. *Xcell Journal*, 3:20–27, 2012.

[13] P. Garcia and K. Compton. Kernel sharing on reconfigurable multiprocessor systems. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 225–232, 2008.

[14] P. Garcia, K. Rupnow, and K. Compton. A reconfigurable computing scheduler optimized for multicore systems. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 107–112, 2010.

[15] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: coordinated scheduling for virtualized accelerator-based systems. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[16] C.-H. Huang and P.-A. Hsiung. Hardware resource virtualization for dynamically partially reconfigurable systems. *Embedded Systems Letters, IEEE*, 1(1):19–23, 2009.

[17] A. Ismail and L. Shannon. Fuse: Front-end user framework for o/s abstraction of hardware accelerators. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 170–177, 2011.

[18] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: first-class gpu resource management in the operating system. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 37–37, Berkeley, CA, USA, 2012. USENIX Association.

[19] Khronos Group Inc. OpenCL project. Website. http://www.khronos.org/opencl/.

[20] R. Kirchgessner, G. Stitt, A. George, and H. Lam. Virtualrc: a virtual fpga platform for applications and tools portability. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA '12, pages 205–208, New York, NY, USA, 2012. ACM.

[21] Nvidia Inc. GRID GPUs . Website. http://www.nvidia.com/object/grid-boards.html.

[22] Oskar Mencer and Stephen Weston. Computational acceleration of credit and interest rate derivatives. Technical report, March 2013. http://www.maxeler.com/media/documents/MaxelerSummaryAccelerationCreditInterestDerivatives.pdf.

[23] C. Plessl and M. Platzner. Virtualization of hardware-introduction and survey. In *ERSA*, pages 63–69, 2004.

[24] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[25] K. Rupnow, W. Fu, and K. Compton. Block, drop or roll(back): Alternative preemption methods for rh multi-tasking. In *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, pages 63–70, 2009.

[26] W. Wang, M. Bolic, and J. Parri. pvfpga: Accessing an fpga-based hardware accelerator in a paravirtualized environment. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–9, Sept 2013.

[27] Y. Wang, J. Yan, X. Zhou, L. Wang, W. Luk, C. Peng, and J. Tong. A partially reconfigurable architecture supporting hardware threads. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 269–276, 2012.

[28] M. A. Watkins and D. H. Albonesi. Remap: A reconfigurable architecture for chip multiprocessors. *IEEE Micro*, 31(1):65–77, 2011.