VIRTFUN: FUNCTION OFFLOAD METHODOLOGY TO VIRTUALIZED ENVIRONMENT

Carlos A Petry and Rodolfo J. de Azevedo

Institute of Computing, University of Campinas, Campinas, Brazil

ABSTRACT

The use of virtual machines (VM) has become popular with substantial growth for both personal and commercial use, especially supported by the progress of hardware and software virtualization technologies. There are several reasons for this adoption like: cost, customization, scalability and flexibility. Distinct domains of application, such as scientific, financial and industrial, spanning from embedded to cloud systems, taken advantage of this kind of machines to meet processing computational demands. However, there are setbacks: hardware handling, resources use, performance and management. This growth demands an effective support by the underlying virtualization infrastructure, which directly affects the hosts' capacity in datacenters and cloud environment that support them. It is evident that the host native processing performs better than VMs, especially when using accelerator devices, where the common solution is to assign each device to a specific VM, instead of sharing it among multiples VMs. Beyond performance issues inside the host, we need to consider the VM performance when using accelerator devices. In this context, it is necessary to provide efficient mechanisms to manage and run VMs which can take advantages of high-performance devices, like FPGAs or even from software resources on the host. To assist this challenge, this paper proposes a methodology to improve communication performance of applications running on the VMs, VirtFun. To do so, we developed a framework able to offload pieces of application's code (vFunction) to host by means of secure data sharing between the application and device. The results achieved in our experiments demonstrated significant acceleration capacity for the guest application vFunction. The speedup reached 340% compared to conventional network execution, reaching maximum slowdown of 2.8% in the worst case and near to 0% in the best case considering the native execution.

KEYWORDS

Virtualization, performance, virtual machine, shared-memory.

1. INTRODUCTION

Cloud computing and virtualization provide increase use of software and hardware computational resources like storage, network, and accelerator devices. Virtual machines (VM) extend support to several domains such as commercial, financial, and scientific spanning from embedded to cloud systems. An impacting factor to execute tasks of client applications falls on the performance of the virtual machine sustained by the host resources. Developers usually employ approaches such as increase of computational resources and distribute processing to meet design requirements and performance issues. However, once VMs rely on the host resources, the greater the amount of existing VMs, the greater will be the pressure on the host machine that support the VMs. Therefore, this significantly increases the processing demand on the host's computing resources. Systems such as processors and memory increase systematically their computational performance and energy efficiency, but resources like network, storage and I/O devices does not have the same progress. Improving virtualization and hardware infrastructures leads to better support for multi-client environment, processing acceleration, resource management, flexibility and security. In addition, optimizing the use of those resources becomes essential to achieve these improvements. In this sense, accelerator resources, especially hardware accelerators, have increased their adoption as an alternative to speed up tasks' execution. A large spectrum of applications can improve performance and energy efficiency by offloading part of their computation to accelerator devices like GPUs and FPGAs. FPGA has significantly increased its use as tasks accelerations. Furthermore, due to the high demand for both scientific and commercial areas, cloud computing has been employing FPGAs as an effective alternative to increase client application performance, especially for tasks that benefits from pipeline.

An important feature for virtualization is the possibility of sharing data between virtual machines and the host system, especially for guest OS running on the same physical machine where the accelerators will be placed. Many approaches allow communication between guest and host OSs, but usually exchange small amounts of data or rely on mechanisms such as OS sockets and network TCP/IP. These methods require copying data between guest and host OSs, causing communication overhead due to the need for serialization and deserialization of data, beside of multiple buffers instances along the data path.

Therefore, this article presents a methodology to allow offload pieces of application's code (vFunction) to accelerator resources inside the host machine by means of the memory sharing binding the application and the resource. Our approach enables to share and manage data exchange efficiently from guest applications located on the same physical machine (host). The proposed approach takes into account security and isolation, providing performance similar to native application execution, besides allowing to develop applications using the standard development tools.

We do not communicate application with the accelerator resources by means of network mechanisms. Instead, the data exchange takes place though regions mapped in memory, where the guest applications will perform I/O operations writing and reading from the mapped memory. Developers can offload pieces of application to vFunctions, seeking to increase performance. We implemented the proposed methodology as a framework that uses several mechanisms like hypercalls, service, mapping and device drivers to enable a communication channel with such functions. The framework has a management and control agent named virtual function daemon (vFunD) which provides a huge administrative and control services. The device driver deals with vFunction both in software and in hardware, acting as an interface between service and the resource (accelerator). The exchange of data between application and vFunction takes place through reading and writing operations in the shared memory region.

The rest of the paper is organized as follow. Section 2 provides a state of art revision of approaches to accelerators resources and high-level software interface efforts. Section 3 presents the proposed methodology and framework implementation explanation. Section 4 provides the experiments carried out to test and validate the effectiveness of the proposed methodology validates by the framework implementation. Section 5 presents the conclusions of the article, future works, and application possibilities of the proposed methodology

2. RELATED WORK

2.1. Studies Related to Accelerators

Putnam et al. [1][2] presented a mechanism to improve datacenters, building reconfigurable FPGA to accelerate large-scale Bing services to face pressure on computing resources. The experiment was evaluated based on throughput and latency compared to the original implementation, exhibited performance gains, increasing the throughput by 95% and reducing latency by 29%, and also an increase of 10% in energy consumption, but the 95% performance gain compensated this difference.

Chen et al. [3] described an effort to include FPGAs in the Cloud to increase performance by accelerating multiple application domains. They mentioned issues to integrate these devices into the cloud: resources abstractions, sharing accelerators, interfacing applications and accelerators and security concerns. To overcome such issues, the authors address four requirements. Abstraction, FPGAs seen as resources, not low-level programmable devices. Sharing, to share FPGAs between multiples clients it is necessary to have a robust isolation mechanism. Compatibility, there are several design workflows for FPGAs which require a common interface to support variety of devices. Security, it is suitable that FPGAs execute independently and in isolation.

Byma et al. [4] proposed a FPGA accelerators integration into cloud OpenStack [5]. They read across multiples physical FPGAs accessed similarly to a virtual machine. They classified FPGAs into three categories: (i) infrastructure, where clients are unaware of them; (ii) appliances, where accelerators run in a box, e.g. Memcached [6][7]; (iii) computing resources, which allow clients to allocate hardware easily. They chose to use computing resources by better serving the cloud diversity.

Fahmy et al. [8] presented an approach to integrate FPGAs into datacenters and cloud to achieve efficiency in performance and power compared to CPUs and GPUs [9][10], targeting streaming applications. The authors faced challenges to integrate FPGAs: (i) dynamic reconfigurable support, (ii) effective communications between multiples accelerators, (iii) optimize FPGA resources over the time, and (iv) easy integration between accelerators and applications. While other works focus on static and monolithic accelerators, they proposed FPGAs as general deployed resources.

Asiatici et al. [11] proposed a methodology and a runtime system to make it easier to implement applications in FPGA, using a high-level API, a hardware and software execution model, and a shared memory model. The framework dynamically manages multiples accelerators allowing designing and mapping multiple accelerators in FPGA, enabling three development levels: DSL, HLS, and RTL. Applications host code run on the host CPU, and offloaded accelerator codes to the FPGA.

Several works proposed to integrate FPGA-accelerated to the cloud infrastructure [3] [4] [12] [14] using OpenStack framework to allow clients to create VMs are able to access FPGAs in a virtualized way. However, their approaches are difficult to integrate in the existing systems. The FPGA virtualization mechanism has to modify the host operating system or insert high overhead. These works do not support communication between virtualized hardware regions. Clients requiring a higher amount of logic might have issues placing their designs because the amount of resources into each virtual FPGA is limited.

Mbongue et al [16] proposed a paravirtualized virtio-based framework to communicate VMs and FPGAs, leveraging partial reconfiguration to run client's hardware VF. They were able to allow designers to request dynamically additional FPGA resources, adding a security layer and allowing runtime reassignment of FPGA resources.

A very common approach to manage FPGAs is by provisioning the FPGAs through OpenStack framework and letting the client connect and program the FPGA through traditional IP or MAC address of the FPGA [4][13][14], allowing the clients to use remote procedure calls or socket connections to communicate with the FPGA.

Some alternatives to Virtio-vsock are virtio-serial and virtual networking proposed by Hajnoczi [17]. The former is a virtual serial device that establishes connections between hosts and guests, but that leads to a limited number of channels. The latter is an approach for guest and host communication using a virtual network, but they can be extremely complex to develop.

2.2. High-Level Interfaces for Accelerators

The design of hardware modules is a highly specialized and time-consuming task. Various efforts, both in academia and industry, have been employed to make hardware development more accessible to non-hardware specialist developers. There are different abstraction levels to design FPGA modules, start from low-level using HDLs like Verilog and VHDL until high-level languages, as C++ and OpenCL [20].

Rise the abstraction level is the main approach to deal with hardware complexity. This methodology provides user-friendly interfaces facilitating the task of handling hardware inherent complexity. To improve development time, it is mandatory to optimize time-consuming FPGA design tasks. One of the efforts to address this challenge is being driven by Intel, the open source framework OPAE [21][22]. OPAE provides a high-level interface between hardware accelerators and client applications through the use of a software stack [23]. OPAE was developed keeping in mind to facilitate the CPU-to-FPGA interaction, especially targeting the Intel HARP [24][25] project that integrates on the same die the Xeon-CPU and FPGA Stratix. The framework provides an API among client applications and FPGA accelerators modules. It also offers an accelerator simulation environment (ASE) [26], a hardware and software co-simulation environment for any Intel FPGA, employed to test and prototype user designs [27]. In addition, the OPAE SDK [28] provides a service-oriented approach to use on user applications [23]. OPAE includes a set of drivers, APIs, and tools to take care of hardware specific actions like discovering, accessing, and reconfiguring the accelerator modules [21].

Although OPAE makes it easier to design FPGA modules, there are some issues to overcome. The OPAE's design flow does not support sharing accelerators among multiple applications [29]. Applying acceleration approaches such as [30][31] can lead to contention in software stack. Although OPAE offers higher abstraction, the designer still needs to have substantial knowledge of the underlying hardware details. Moreover, there is a lack of intrinsic support to deploy multiple acceleration modules in the same design, leading to manually instantiate each accelerator in a set of modules [27].

3. VIRT-FUN: ACCELERATOR OFFLOAD METHODOLOGY

This section presents details of the proposed methodology and the framework implemented to validate this methodology. We explain how a client application, running on a virtual machine on the same physical host (co-located) make use of vFunction.

3.1. Layers Assessment

Figure 1 shows the infrastructure considering four abstraction layers, from the client application on top to the hardware on bottom. Solid lines represent the flow related to data, whereas dashed lines indicate the flow related to control issues.

At topmost, the App Layer represents the client applications running on virtual machines, where each of them can execute several applications. Furthermore, there may be multiple instances of VMs running multiple applications each one at the same time. In this layer, client develop and submit their applications with the possibility to offload pieces of the code to the vFunctions on the co-located host. As usual, developers can write code in a variety of languages supported by the platform.

In the next layer, VMOS comprises a number of running VMs (VM1, VM2) and supporting client applications (depicted as: 1A, 1B, 2A, 2B) from a number of clients (e.g.: Cli1, Cli2). Each VM executes applications of only one client, and each client can launch a number of applications as they need or the platform supports. Each VM has a memory region to share data with user-defined vFunctions, however each client application maintains their data in a distinct region.

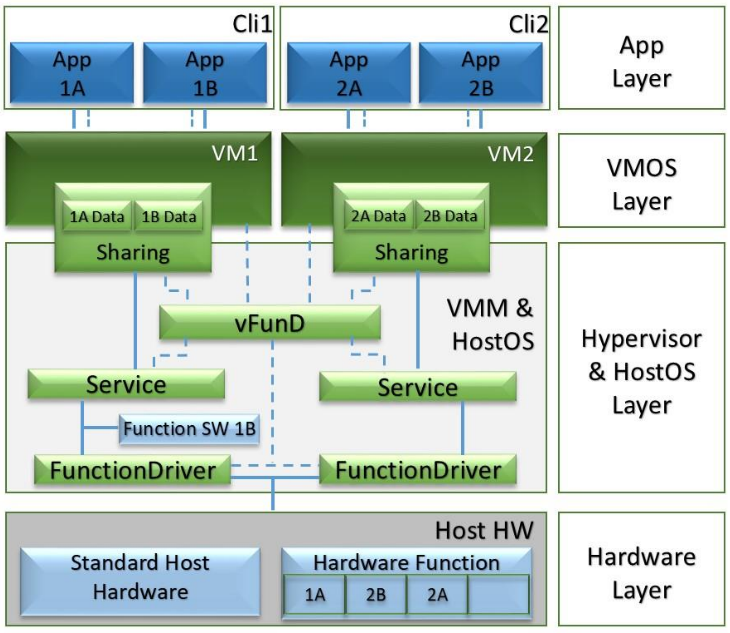


Figure 1. Overall layered framework view.

In the third layer, the VMM & HostOS integrates the main components of the framework. If a client decides to offload the vFunction to software, a version of that function will stay in place of the hardware version. The management software, vFunD, performs the administrative and security tasks by registering, launching, controlling and releasing all vFunctions. The manager will also provide security and isolation between all shared regions. The service component runs similarly to a daemon and is responsible for handling communication requests to/from the vFunction driver. In addition, it takes care of status conditions and command actions built-in in the communication protocol. FunctionDriver is a high-level driver responsible for supporting the low-level communication with hardware, or software version, communicating with the OS device-driver where the vFunction was instantiated. In case of hardware vFunction, the service uses the FunctionDriver to interact with the low-level device-driver supplied by the hardware manufacturer, included in the board support package (BSP) library.

The lower layer, Hardware, represents the physical host, e.g., CPU, memory and storage, etc, together with the underlying devices, labeled as Hardware Function. In the context of this work, we target especially the FPGAs devices.

3.2. Communication Flow

In this subsection, we present the execution flow of the proposed methodology applied to the framework, starting from the guest application and following until reach the vFunction on the host. Figure 2 shows a timed diagram consisting of three columns separated by two vertical lines. The leftmost column indicates the actions taken by the client in the context of the guest application, where the right column is responsible for actions belonging to the framework on the host. The central column shows the dynamics of events generated on both sides of the processing.

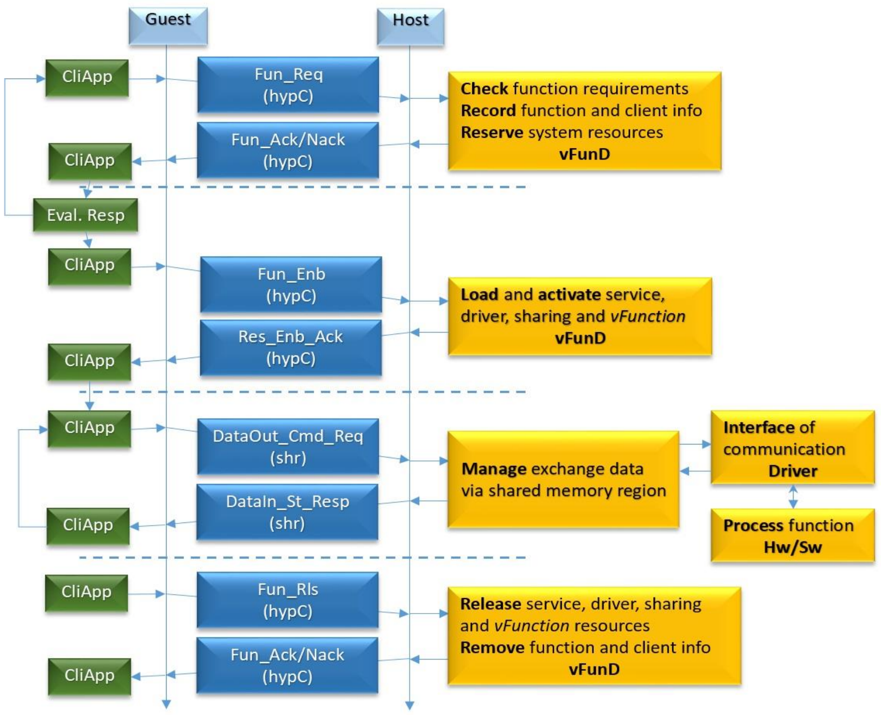

Guest
Host
CliApp
Fun_Req (hypC)
Check function requirements
Record function and client info
Reserve system resources
vFunD
Fun_Ack/Nack (hypC)
Eval. Resp
Fun_Enb (hypC)
Load and activate service, driver, sharing and vFunction
Res_Enb_Ack (hypC)
DataOut_Cmd_Req (shr)
DataIn_St_Resp (shr)
Manage exchange data via shared memory region
Interface of communication Driver
Process function Hw/Sw
Fun_Rls (hypC)
Release service, driver, sharing and vFunction resources
Remove function and client info

Figure 2. Time diagram showing the execution flow of the framework.

The execution follows the client-server model, thus the client takes all actions and the server returns the responses. Briefly, the sequence of events happens in four phases as follows.

3.2.1. Negotiation Phase

The client application requires that part of its computation to be executed in a vFunction on host. To do so, it invokes a standard systemcall, which in turn invokes a new hypercall implemented into host kernel to respond to this request, this the unique invasive method employed by the framework. The hypercall forwards the demand to the device resource manager (vFunD), that checks all requirements related to the request, such as resource availability, processing capacity, credential, etc. If successful, it reserves the suitable resources and records the request details, like client credential (clientID), application and function identity (AppID, functionID), and vFunction metadata (functionLoc). At the end of this phase, vFunD returns the request status including ack or nack conditions along with status reason.

3.2.2. Activation Phase

In this phase, the application analyses the vFunD return, and if function request becomes available, the application can proceed to next phase. Otherwise, the client application can request the same function in software, instead of in the hardware. However, if the reason for failure has been unrecognized resources, possibly this would prevent the execution to continue. On an ack return, the application issues another systemcall, this time requesting the vFunction activation. Again, vFunD processes the systemcall and load the service, driver, vFunction resources, and activates the shared memory region. Moving forward, vFunD updates the request status and launches the vFunction making it available. Finally, it returns the request status including ack or nack conditions along with the status response. On the guest side, once the vFunction is available the application advances to the next phase, where the vFunction computation will be exploited.

3.2.3. Computation Phase

The computation phase allows the application to exchange data with the vFunction in the host. Communication transfers data through the shared memory region in accordance to the commands defined in the aforementioned protocol and check the state conditions emitted by the service. The command set sends application’s data to the vFunction to processes them in the host, while the command get takes the result of the computation. The command rst allows the application clear data inside vFunction if the project foresees such functionality. There are three essential status conditions for service: wait, when service detects that the vFunction is ready to process new requests; busy, indicates that the device is busy and performing other requested computations; done, stating that the device has finished a requested computation.

3.2.4. Release Phase

In this phase, the application notifies the vFunD to release all resources and information related to the ordered vFunction, also through a systemcall. vFunD processes the order calling a hypercall to stop the service, unload driver and vFunction resources and, deallocate the shared memory region. In addition, vFunD remove administrative entries such as credential, resources and any other information related to the vFunction previously instantiated. Finally, it returns the request status including ack or nack conditions along with the status reason. On the application side, the release confirmation enables the application to remove the vFunction.

3.3. Implementation and Data Flow

In virtualization, the guest OS runs over the same physical machine (co-located) where the host and hypervisor are also running. However, the guest OS may need to communicate with a resource that is located on the same physical machine or on a remote machine. Resources locally available can provide some desirable advantages and features. Locally provided resources make it possible to implement a file system shared between gest and host OSs. This approach is under development, but in the final stage, through of the VirtioFS project [18][19], which should be part of the Linux kernel mainline soon. Based on this approach, we can implement the data sharing between the guest application and the vFunction on the host through a shared resource by means of a file system mapped to a shared memory region. Through a memory mapped I/O region, the application on guest and the service on host can exchange data using simple variable assignments. This mechanism makes it possible to provide a local data flow mechanism without face the overhead of traditional communication, as the TCP/IP.

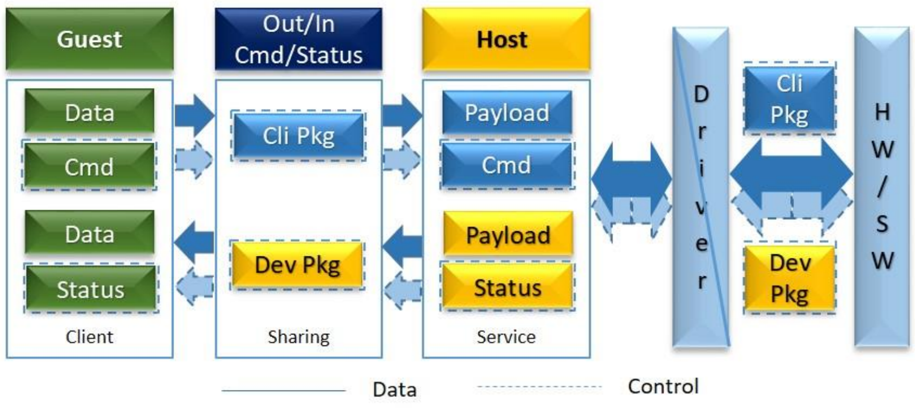


Figure 3. Data flow overview of the framework.

Within the application depicted to the left of Figure 3 (Client, green part), the developer accesses the shared resource by opening files via the traditional open() function and maps them into memory, assigning the returned address to a data structure by the mmap() function. We chose to separate the mapping in two parts: (a) dataout, used to send data for processing together with the client command (set, get, rst); (b) datain, used as a data receiver containing the result of computation together the state condition. The Figure 4 shows a code snippet of the above-mentioned dynamic.

(a) opening and mapping data sharing:

```
fd_out = open(file_out, O_RDWR | O_CREAT | O_SYNC);
dataout_srv = (argout_t*) mmap(NULL, BUFFER_SIZE,
    PROT_READ | PROT_WRITE, MAP_SHARED, fd_out, 0);
fd_in = open(file_in, O_RDONLY | O_SYNC);
datain_srv = (argin_t*) mmap(NULL, BUFFER_SIZE,
    PROT_READ, MAP_SHARED, fd_in, 0);
```

(b) assigning values to data share:

```
dataout_srv->cli_data.field1 = value1;
dataout_srv->cli_cmd = e_set;
```

(c) reading data sharing values:

```
while (datain_srv->srv_st == e_req_wait);
```

Figure 4. Dataflow code snippet.

Still in the center of Figure 3, the service (yellow host column) executes the dynamic similar to that performed by the client, regarding access to shared data. However, shared data is carried in reverse, and client commands are identified and manipulated to grant the management and communication with the vFunction resource.

```
<filesystem type='mount' accessmode='passthrough'>
  <driver type='virtiofs' queue='1024'/>
  <binary path='/path_to_virtiofs_daemon/virtiofsd'>
    <cache mode='always'/>
    <lock posix='on' flock='on'/>
  </binary>
  <source dir='/path_on_host/vm-share/app-share'/>
  <target dir='app-share-ID'/>
</filesystem>
```

Figure 5. Guest OS directives to enable the data sharing structure.

The central part of Figure 3 (dark blue column) corresponds to the shared data mechanism based on VirtioFS configured according to the rules outlined in Figure 5. To work properly, it is necessary to map a directory on the host OS inside the guest OS file system. There are different ways to configure this mapping, for this work we chose to use the XML writing directives describing the guest VM for the hypervisor in this file. A priori, one can attach the mount point into any directory path inside the guest. However, it is suggested to map the shared resource within the user's home or in the /var path. Permissions and ownership of files inside of hierarchy will be defined by the guest operating system.

The guest mount point will be restricted to the host directory and the directories below, in other words the guest sees the shared resource as the root mount point. It is not possible to escape from this hierarchy due to two security mechanisms: sandbox that implements kernel namespace for processes, and seccomp that restricts system calls invoked by the processes. vFunction provides the computation resources for guest applications through the driver and the specialized hardware or software on the host, as shown at the rightmost Figure 3 (light blue part).

Another choice made for this work was to split the driver functionality into two parts and, denoted in the driver box in Figure 3, by the diagonal line over component. This led to the device driver refactoring in two parts: (i) device critical functions, such as highly-device dependent features (low-level driver); (ii) remaining non-critical (highest level) functionalities (high-level driver). Devices manufacturers, here especially FPGAs, make their low-level drivers available through the platform's BSP. Design and synthesis tools, such as Quartus and Vivado, automatically produce high-level headers interfaces to integrate the service back-end as low-level interfaces. The high-level driver consists of three primary functionality: (i) device_driver_init, which registers and enables the driver making it accessible; (ii) device_driver_exit, which disables and removes the driver instance freeing the resources on the host OS; (iii) driver_function, which implements the communication interface with the low-level manufacturer driver. If client chooses to offload functions in software, the driver will communicate directly with that vFunction version carried out as a daemon.

```
/var
 ├─ vFun
     ├─ admin      ├─ service        ├─ vmcli
     │   └── vfm   │  └─ cli1        │  └─ cli1
                   │       └─ app1A  │     └─ app1A
                   │       └─ app1B  │     └─ app1B
                   │  └─ cli2        │  └─ cli2
                   │       └─ app2A  │     └─ app2A
                   │       └─ app2B  │     └─ app2B
```

Figure 6. Guest OS directives to enable the data sharing structure.

Figure 6 shows the file and directory host OS infrastructure provided to support the vFucntion's functionalities. The root hierarchy (/var) was split into three branches: administrative components (admin), service provision (service), and shared resources (vmcli). Administrative components branch will contain the vFunD management software indicated in the Hypervisor and HostOS layer of Figure 1. It controls and implements all requests related to applications vFunction, which offload pieces of its code to specialized devices on the host.

The service provision branch holds the interface software acting as service for each vFunction request. Inside this branch, each client will have its own directory named by the clientID that will hold one subdirectory for each client's requested function, named according the vFunctionID. Within each subdirectory there will be a service daemon executable to interface the application and the vFunction. Each service daemon controls the communication between application and vFunction resource, attached to the high-level driver of that resource. Finally, shared resource branch will have one directory for each client named by the clientID, similar to the service provision branch, but that will become the root mount point for each client inside the guest VM. Within that mount point, there will be a subdirectory for each function requested by the client, named by the vFunction ID. This directory will be linked to the shared resource and will become the shared memory region. The daemon service executable will consider this local to enable the structures dataout and datain to store and access data.

4. Experimental Results

4.1. Environment

We implemented a prototype to validate the proposed methodology to offload VM virtualized functions to the host. We used the following hardware and software setup. HW: Intel Core i7 processor, 16GB of RAM and a SSD storage drive of 480GB. SW: GNU/Linux Ubuntu server 20.04 with kernel 5.4.0 and KVM support enabled, hypervisor QEMU 4.2.50 and Libvirt library 6.3.0. We used both software from sources in order to make available the latest functionalities, especially the VirtioFS support, not yet available in the mainline packages.

4.2. Methodology

Currently, we validated the framework by offloading two distinct vFunctions to software version: Factorial and Fibonacci. Although the primary goal is to virtualize functions in hardware, the software implementation is sufficient to validate the methodology, since such implementations are irrelevant from the framework point of view. We compared the behavior of three domains of execution. Host-Host: vFunctions are invoked and executed both on the host (native), this is the reference domain. Guest-Host: vFunctions are invoked inside the VM and executed on the host (virtualized), the target domain. Guest-Net: vFunctions are invoked by the VM application through the network and executed on the host (network). We conducted the experiment execution considering four parameters:

- **Domain**: representing the three aforementioned communication approaches, Host-Host, Guest-Host and Guest-Net.
- **Instance**: which comprises a round of calculation characterized by two arguments: repetition, the number of calculation replay, and iteration, the amount of calculation invocations.
- **Time**: the time spent by each instance execution, obtained by the arithmetic mean of five iterations.
- **Value**: number sent to the vFunction fixed as 51 for Factorial and 93 for Fibonacci, chosen for generating the largest result fitted into a 64-bit variable.

4.3. Evaluation

Tables 1 and 2 present the execution times of the Factorial and Fibonacci vFunction instances, measured in seconds. The first column indicates the names of the function and the three domains of execution, while the additional columns show the values for five instances executed in the experiment, referenced as: repetitions x iterations. We start processing instances on Host-Host domain, recording the execution times for the reference domain (native), shown in the second row of both tables. Here, the execution time increases as the number of iterations grows, which it is expected since each invocation of the function leads to an additional overhead. For example, comparing the execution time for the Factorial vFunction on Host-Host domain, the time increases by 58.7%, considering the smallest and largest results (invocations 1:2k). We expected this situation since the more invocations, the greater is the overhead to handle them. We also measured instances executions for Guest-Host and Guest-Net domains, recording results for Factorial and Fibonacci vFunctions in that tables, shown by the next two rows.

Table 1. Results of the execution times of the instances for the factorial vFunction considering the three domains.

Factorial	500M x 1	50M x 10	5M x 100	500K x 1k	250K x 2k
Host-Host	**16,40**	**16,45**	**16,79**	**20,87**	**26,02**
Guest-Host	**16,41**	**16,57**	**16,81**	**21,44**	**26,54**
Guest-Net	**15,98**	**16,52**	**20,84**	**65,40**	**114,81**

Evaluating instances execution with 1 and 10 iterations (500Mx1 and 50Mx10 columns), we observed that the execution time remains virtually the same for all domains, around 16s for Factorial and 19s for Fibonacci. This indicates that the system resources are able to handle properly the invocations for this workload. On the other hand, for instances executions starting from 100 iteration this scenario changes, reaching close to 619%, considering the lower and higher results for the Guest-Net domain (invocations 1:2k). However, the same does not happen if we compare the Host-Host domain with the Guest-Host, where the increase reached 61.7%, considering the same invocations case (1:2k).

Table 2. Results of the execution times of the instances for the Fibonacci vFunction considering the three domains.

Fibonacci	500M x 1	50M x 10	5M x 100	500K x 1k	250K x 2k
Host-Host	**19,68**	**19,78**	**20,03**	**23,92**	**29,62**
Guest-Host	**19,69**	**19,80**	**20,08**	**24,87**	**30,13**
Guest-Net	**19,18**	**19,79**	**26,17**	**70,99**	**118,16**

Figures 7 and 8 present a graphical evolution of all instance's executions for every domain. One can see that the leftmost two bars (500Mx1 and 50Mx10 instances) remains almost the same indicating that executions in the three domains, represented by colorized bars, have the same execution time, therefore are equivalent in performance. However, for the executions of the rightmost three sets of bars (5Mx100, 500Kx1K and 250Kx2k instances) the Guest-Net domain stands out for its variation, reaching about 340% in the case of Factorial. This does not happen with other two domains (Host-Host and Guest-Host) that grows in equal proportions. This demonstrates that the Guest-Net approach is not scalable, which is evidenced by the non-linear growth of the gray bars of both figures.

Comparing the execution time growth between the Guest-Host and Host-Host domains, one can see that it behaves in a proportional rate, as can be seen by the shapes sequence of blue and red bars shown in Figures 7 and 8. We confirm this condition in the 2nd and 3rd lines in Tables 1 and 2, where the values, in both lines, increase virtually in the same proportion. This demonstrates the effectiveness of the proposed methodology, since the growth of time to run instances on the Guest-Host domain (virtualized) behaves in a proportional way to the Host-Host reference domain (native).

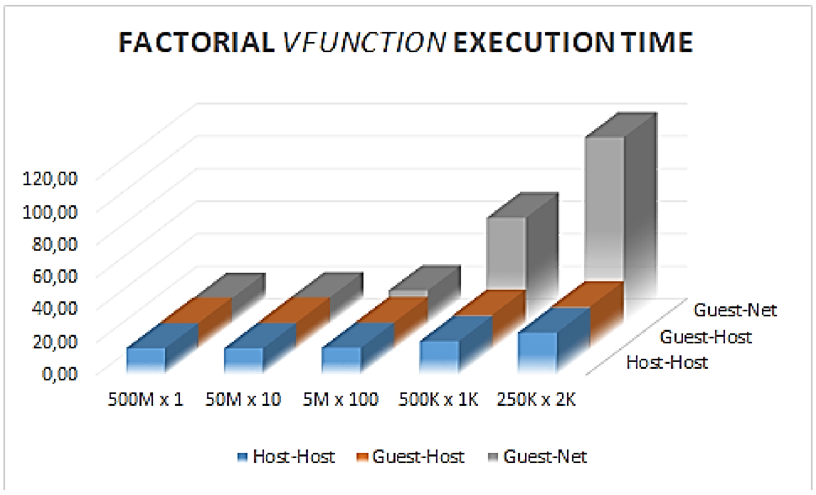


Figure 7. Execution time of instances for all domains of factorial vFunction, in seconds.

We can demonstrate that the vFunctions execute with the equal performance if we compare the Guest-Host domain and Host-Host (native) domain. Furthermore, the methodology effectiveness can be observed through Figure 9, which shows the executions of the instances compared as percentages for both vFunctions applications. The executions percentages in the Host-Host domain as well as the Guest-Host tend to zero, depicted by the labels GH-HH of Figure 9.

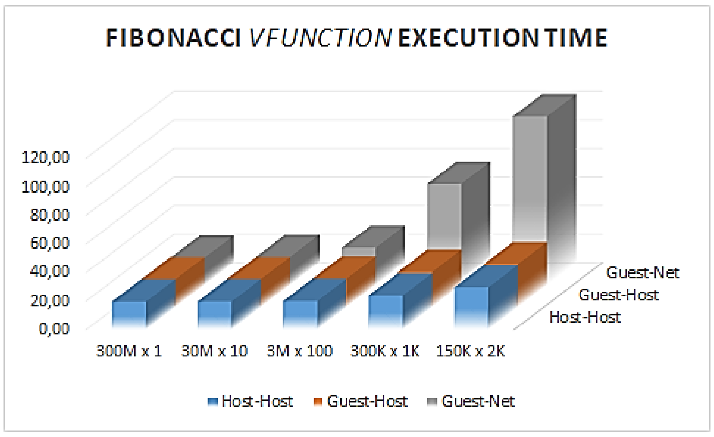
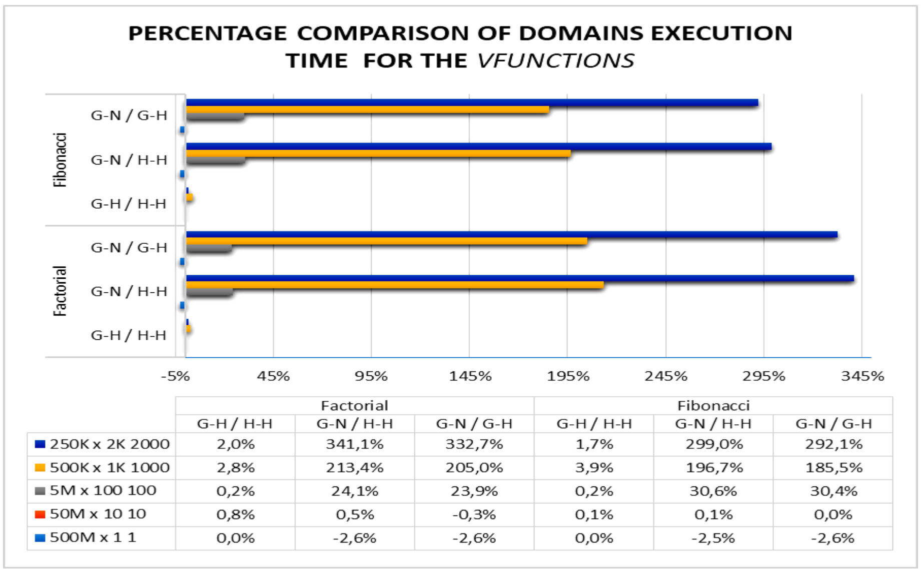


Figure 8. Execution time of instances for all domains of Fibonacci vFunction, in seconds.

On the other side, labels containing the GN (Guest-Net) label present high (non-linear) execution time growth, indicating that for applications with high demands the increase time becomes a bottleneck. Since the performance of domains Guest-Host and Host-Host is proportional, the virtualized domain represents a substantial speedup mechanism compared to the traditional domain (network) and an optimum choice to offload vFunctions on the co-located host and administrate the interaction between them.

Note: H, G and N letters in Figure 9 represent Host, Guest and Network, respectively. In addition, G-H/H-H, and other equivalent expressions, represent the percentage calculated between the instance's values of the Guest-Host domain and the Host-Host.



	Factorial			Fibonacci		
	G-H / H-H	G-N / H-H	G-N / G-H	G-H / H-H	G-N / H-H	G-N / G-H
■ 250K x 2K 2000	2,0%	341,1%	332,7%	1,7%	299,0%	292,1%
■ 500K x 1K 1000	2,8%	213,4%	205,0%	3,9%	196,7%	185,5%
■ 5M x 100 100	0,2%	24,1%	23,9%	0,2%	30,6%	30,4%
■ 50M x 10 10	0,8%	0,5%	-0,3%	0,1%	0,1%	0,0%
■ 500M x 1 1	0,0%	-2,6%	-2,6%	0,0%	-2,5%	-2,6%

Figure 9. Execution times percentage comparison for each vFunctions for all domains.

5. CONCLUSION

In this article, we present a methodology to allow pieces of an application's code (vFunction) to be offloaded to acceleration hardware resources. Alternatively, this code can also be delegated to a software version on the same host, that was our choice to validate the proposed methodology. We developed a framework to validate the methodology and conducted experiments to validate the effectiveness of the proposal. The experiment made use of two vFunction and five variations of executions, performed over three domains. We provided a noninvasive solution (except for a new hypercall) in which the client can use the standard development tools to implement applications. The experiments demonstrated that, for applications with about 100 iterations or more, the speedup of execution time reach substantial gains, over 340% for 2k iterations, comparing the Guest-Host domain and the Guest-Net. The experiments also indicated that the acceleration increases in a linear way when we compare the native and virtualized domains. Therefore, the proposed methodology proved to be an excellent option to offload vFunctions and administrate the guest-host interaction getting good performance results.

Based on the promising results, future works consider integrating the framework with hardware devices, providing the underlying backend especially for FPGAs, since these devices have recent improvements and new useful features. Additionally, we intend to integrate the framework together with the backend in the cloud, providing a complete multi-tenant computing infrastructure to enable the vFunction accelerator to cloud VMs.

ACKNOWLEDGEMENTS

This work is supported by the Sao Paulo Research Foundation (FAPESP) (2013/08293-7), CAPES (2013/08293-7), and CNPq (fellowships 438445/2018-0, 309794/2017-0).

REFERENCES

[1] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In 2014 ACM/IEEE 41st International Symposium on Computer Architecture(ISCA), pages 13-24, June 2014.

[2] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong,P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. IEEE Micro, 35(3):10-22, May 2015.

[3] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling fpgas in the cloud. In Proceedings of the 11th ACM Conference on Computing Frontiers, CF ’14, pages 3:1-3:10, New York, NY, USA, 2014. ACM.

[4] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines, pages 109-116, May 2014.

[5] OpenStack Overview. https://www.openstack.org/software/, 2020. Online: accessed 21-Feb-2020.

[6] Brad Fitzpatrick. Distributed caching with memcached. Linux J., 2004(124):5, August 2004.

[7] Brad Fitzpatrick. Memcached, a distributed memory object caching system. https://memcached.org/, 2020. Online: accessed 24-Feb-2020.

[8] S. A. Fahmy, K. Vipin, and S. Shreejith. Virtualized fpga accelerators for efficient cloud computing. In 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), pages 430-435, Nov 2015.

[9] S. Asano, T. Maruyama, and Y. Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In 2009 International Conference on Field Programmable Logic and Applications, pages 126-131, Aug 2009.

[10] S. Kestur, J. D. Davis, and O. Williams. Blas comparison on fpga, cpu and gpu. In 2010 IEEE Computer Society Annual Symposium on VLSI, pages 288-293, July 2010.

[11] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne. Virtualized execution runtime for fpga accelerators in the cloud. IEEE Access, PP(99):1-1, 2017.

[12] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling fpgas in hyperscale data centers," in Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), 2015 IEEE 12th Intl Conf on. IEEE, 2015, pp. 1078–1086.

[13] J. Weerasinghe et al., "Network-Attached FPGAs for Data Center Applications," in FPT, Dec 2016.

[14] N. Tarafdar, N. Eskandari, T. Lin, and P. Chow, "Designing for fpgas in the cloud," IEEE Design & Test, vol. 35, no. 1, pp. 23–29, 2018.

[15] N. Tarafdar et al., "Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center," in FPGA '17. ACM, 2017

[16] J. Mandebi Mbongue, F. Hategekimana, D. Tchuinkou Kwadjo and C. Bobda, "FPGA Virtualization in Cloud-Based Infrastructures Over Virtio," 2018 IEEE 36th International Conference on Computer Design (ICCD), Orlando, FL, USA, 2018, pp. 242-245, doi: 10.1109/ICCD.2018.00044.

[17] S. Hajnoczi, "virtio-vsock Zero-configuration host/guest communication". https://vmsplice.net/~stefan/stefanha-kvm-forum-2015.pdf. Online: accessed 05-Mar-2020.

[18] Red Hat, Inc, "virtiofs: virtio-fs host<->guest shared file system". https://www.kernel.org/doc/html/latest/filesystems/virtiofs.html. Online: accessed 20-Feb-2020.

[19] Virtio-fs WebSite, "Virtio-fs - shared file system for virtual machines". https://virtio-fs.gitlab.io/. Online: accessed 15-Feb-2020.

[20] M. Gémieux, M. Li, Y. Savaria, J. David and G. Zhu, "A Hybrid Architecture With Low Latency Interfaces Enabling Dynamic Cache Management," in IEEE Access, vol. 6, pp. 62826-62839, 2018.

[21] Intel Open Source, "OPAE-Open Programmable Acceleration Engine". https://01.org/OPAE. Online: accessed 20-May-2020.

[22] Intel Open Source, "OPAE repository". https://github.com/OPAE. Online: accessed 20-May-2020.

[23] P. Colangelo, E. Luebbers, R. Huang, M. Margala and K. Nealis, "Application of convolutional neural networks on Intel® Xeon® processor with integrated FPGA," 2017 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, 2017, pp. 1-7.

[24] Intel Look Inside, "IvyTown Xeon + FPGA: The HARP Program". https://cpufpga.files.wordpress.com/2016/04/harp_isca_2016_final.pdf. Online: accessed 20-May-2020.

[25] Y. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, P. Wei, "A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms", 2016 ACM Design Automation Conference (DAC), Austin, TX, 2016, pp. 1-6.

[26] Intel Corpoation, "Intel Accelerator Functional Unit Simulation Environment Quick Start User Guide (ASE)". https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qs-ase.pdf. Online: accessed 20-May-2020.

[27] L. Bragança, F. Alves, J. Penha , J. Penha, G. Coimbra, R. Ferreira, J. Nacif, " Simplifying HW/SW integration to deploy multiple accelerators for CPU-FPGA heterogeneous platforms", 2016 ACM International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), Samos, Greece, 2018, pp. 97-104.

[28] OPAE SDK, "OPAE SDK source code repository". https://github.com/OPAE/opae-sdk. Online: accessed 20-May-2020.

[29] S. Rezaei, K. Kim and E. Bozorgzadeh, "Scalable Multi-Queue Data Transfer Scheme for FPGA-Based Multi-Accelerators," 2018 IEEE 36th International Conference on Computer Design (ICCD), Orlando, FL, USA, 2018, pp. 374-380.

[30] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, " RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators," 2012 ACM ACM Transactions on Reconfigurable Technology ans Systems, 8,4, , pp. 1-23.

[31] M. Vesper, D. Koch, K. Vipin and S. A. Fahmy, "JetStream: An open-source high-performance PCI Express 3 streaming library for FPGA-to-Host and FPGA-to-FPGA communication," 2016 26th International Conference on Field Programmable Logic and Applications (FPL), Lausanne, 2016, pp. 1-9.

AUTHORS

Carlos A Petry graduated in Computer Science at the University of Passo Fundo in 1990. He completed his master's degree in Computer Science at the Pontifical Catholic University of Rio Grande do Sul in 2009. He is currently pursuing his doctorate at the State University of Campinas. Professor at the Federal Institute of Rio Grande do Sul in the Computer Science course in the areas of Computer Architecture, Digital Systems, Computer Theory and Compilers, oriented undergraduate students.

Rodolfo J de Azevedo graduated in Computer Engineering from the Federal University of Espírito Santo in 1998, PhD in Computer Science from the State University of Campinas in 2002 and post-doctorate from the University of Washington - USA in 2010. Has a scholarship in Research Productivity from CNPq level 2 Associate professor at the State University of Campinas - UNICAMP, guiding students of the master's and doctorate in Computer Science in the areas of Computer Architecture, Dedicated Systems, Hardware Design and Use of Technology in Education.