A Hypervisor for Shared-Memory FPGA Platforms

Jiacheng Ma* Gefei Zuo* Kevin Loughlin* Xiaohe Cheng§ Yanqiang Liu†
Abel Mulugeta Eneyew‡ Zhengwei Qi† Baris Kasikci*
*University of Michigan §Hong Kong University of Science and Technology
†Shanghai Jiao Tong University ‡Addis Ababa Institute of Technology

Abstract

Cloud providers widely deploy FPGAs as application-specific accelerators for customer use. These providers seek to multiplex their FPGAs among customers via virtualization, thereby reducing running costs. Unfortunately, most virtualization support is confined to FPGAs that expose a restrictive, *host-centric* programming model in which accelerators cannot issue direct memory accesses (DMAs). The host-centric model incurs high runtime overhead for workloads that exhibit pointer chasing. Thus, FPGAs are beginning to support a *shared-memory* programming model in which accelerators can issue DMAs. However, virtualization support for shared-memory FPGAs is limited.

This paper presents OPTIMUS, the first hypervisor that supports scalable shared-memory FPGA virtualization. OPTIMUS offers both spatial multiplexing and temporal multiplexing to provide efficient and flexible sharing of each accelerator on an FPGA. To share the FPGA-CPU interconnect at a high clock frequency, OPTIMUS implements a multiplexer tree. To isolate each guest's address space, OPTIMUS introduces the technique of page table slicing as a hardware-software co-design. To support preemptive temporal multiplexing, OPTIMUS provides an accelerator preemption interface. We show that OPTIMUS supports eight physical accelerators on a single FPGA and improves the aggregate throughput of twelve real-world benchmarks by 1.98x-7x.

CCS Concepts • **Hardware Reconfigurable logic and FPGAs**; • **Software and its engineering Virtual machines**.

Keywords OPTIMUS, FPGA, Virtualization

ACM Reference Format:
Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. 2020. A Hypervisor for Shared-Memory FPGA Platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3373376.3378482

ACM ISBN 978-1-4503-7102-5/20/03...$15.00
https://doi.org/10.1145/3373376.3378482

1 Introduction

Field Programmable Gate Arrays (FPGAs) allow users to significantly accelerate custom workloads, including those of machine learning [60–62, 85, 87], compression [57], scientific computing [23], database operations [52, 62], and graph analytics [10, 89]. As the set of data center workloads changes over time, cloud providers can reconfigure their FPGAs into different accelerators, making FPGAs a cost-effective and flexible alternative to ASICs [16, 63].

Considering the high non-recurring engineering cost [38] of hardware design and the fact that most cloud application developers are software programmers, cloud providers such as Amazon and Microsoft configure their FPGAs into popular accelerators, which the providers then make available for customer use [6, 50].

As with other hardware devices, cloud providers desire the ability to multiplex their FPGAs among different customers via virtualization, thereby increasing resource utilization and return on investment (ROI) [37, 71]. Although multi-tenant FPGA hypervisors and operating systems exist [15, 18, 21, 37, 40, 53, 55, 72–74, 86], these solutions are restricted to FPGA platforms that expose a *host-centric* programming model, as opposed to a *shared-memory* model.

The key difference between host-centric and shared-memory FPGA programming models is whether or not accelerators can issue direct memory accesses (DMAs, via which an I/O device obtains data from system memory). In host-centric models, the host issues all DMAs via a CPU-configured DMA engine, which passes the accessed data to the necessary accelerator; the accelerators themselves cannot issue DMAs. Most FPGA manufacturers [7, 68, 81] adopt this programming model. Unfortunately, the host-centric model cannot efficiently support applications that exhibit pointer chasing (e.g., graph processing [76] and database acceleration [62]), as such applications require repeated communication between the CPU and FPGA to coordinate each DMA. In particular, the software programmer must either 1) initiate multiple data transmissions separately and sequentially, or 2) marshal the data every time before transmission, both of which hurt performance.

To overcome the performance penalties of the host-centric programming model, emerging FPGAs are alternatively exposing a lighter, more flexible shared-memory programming model [9, 25, 27, 66]. Under this new model, each accelerator can issue its own DMAs and shares an address space with a process on the CPU. The CPU is merely responsible for providing the accelerator with a pointer to its initial input data. Upon receiving the pointer, the accelerator can issue the initial and subsequent DMAs without CPU intervention. As we demonstrate in §2.1, the shared-memory model can outperform the host-centric model by 37%–85% in a virtualized environment.

Unfortunately, virtualizing system memory on shared-memory FPGA platforms is challenging. In particular, because both the CPU and FPGA can directly access system memory, virtualization solutions must provide consistent views to applications on the CPU and accelerators on the FPGA. For instance, if a software process updates a page's data/metadata, these changes must be immediately visible to its corresponding accelerator, and vice-versa.

Furthermore, while SR-IOV [43] (i.e., hardware-assisted IO virtualization) provides a method of isolating virtual DMAs on *PCIe* links, shared-memory platforms can expose an interface that encapsulates both a PCIe link and a UPI link (e.g., Intel HARP [25]). Thus, on such platforms, SR-IOV does not provide a comprehensive solution to virtual DMA isolation. Additionally, for the past five years, shared-memory platforms have been unable to support more than one VF per FPGA [25, 27], limiting SR-IOV's scalability on these platforms.

In this paper, we introduce OPTIMUS, the first scalable hypervisor that virtualizes shared-memory FPGAs. Deployed by cloud providers, OPTIMUS can configure a single FPGA into well-isolated accelerators, simultaneously accelerating a variety of jobs and improving resource utilization.

OPTIMUS targets a use case in which cloud providers configure FPGAs as a set of popular accelerators for their customers (e.g., the accelerator libraries/registries of Amazon F1 [6] and others [18, 37]). Notably, OPTIMUS does *not* aim to virtualize an FPGA's reconfiguration capabilities, opting instead to schedule VMs on FPGAs pre-configured with the necessary accelerator(s). Such a model is desirable in a cloud setting, as it 1) avoids the high performance overheads—and therefore, revenue losses—of reconfiguration during accelerator context switches, and 2) still allows cloud providers to reconfigure their *physical* FPGAs as customer needs change over time.

OPTIMUS virtualizes shared-memory FPGAs via a composition of spatial multiplexing and temporal multiplexing. *Spatial multiplexing* partitions the physical FPGA into multiple accelerators that can be individually controlled by different VMs [15, 18, 21, 37, 41, 72, 74]. *Temporal multiplexing* then oversubscribes these accelerators—multiple VMs take turns running atop a fixed-configuration accelerator [16, 73]. To support temporal multiplexing, OPTIMUS offers a preemption interface for accelerator design, such that it can instruct virtual accelerators to swap their state to/from system memory on a context switch.

OPTIMUS is implemented atop Intel Skylake HARP [25], but its design can be generalized to different shared-memory FPGA platforms. OPTIMUS efficiently overcomes the DMA isolation limitations of existing shared-memory FPGAs with a virtualization technique called *page table slicing*. Page table slicing is inspired by prior software-only techniques on isolating DMAs [70, 78], but is instead implemented as a generic hardware-software co-design to provide virtualization independent of specific accelerator configurations. Using page table slicing, OPTIMUS configures the FPGA to include a *hardware monitor*, which assists in partitioning a single IO page table among all guests without incurring IO page table context switching overhead.

OPTIMUS spatially multiplexes up to eight unique physical accelerators and improves the aggregate throughput of twelve real-world benchmark workloads by 1.98x-7x. Additionally, OPTIMUS's hardware monitor occupies less than 7% of FPGA resources. Finally, OPTIMUS stringently enforces real-time bandwidth sharing policies for both spatially- and temporally-multiplexed accelerators.

In summary, this paper makes the following contributions:

- We design OPTIMUS, the first scalable hypervisor to offer virtualization support for shared-memory FPGAs, using both spatial multiplexing and temporal multiplexing to provide efficient, fair, and flexible sharing of individual accelerators on an FPGA.
- We introduce a hardware-software co-design for IO virtualization—*page table slicing*—that isolates each virtual accelerator's DMAs via a combination of hypervisor and on-FPGA support.
- We provide an interface to support the inclusion of preemption capabilities in accelerator design.

2 Background

Field Programmable Gate Arrays (FPGAs) are chips that can be configured (and reconfigured) into custom circuits (e.g., accelerators). FPGA developers often use hardware description languages such as Verilog [69] and VHDL [12] to describe their circuit designs. A *synthesizer* program translates these designs into native FPGA *bitstreams* (i.e., binaries).

In the rest of this section, we give detailed background on FPGA programming models as well as FPGA virtualization. We focus on FPGAs designed to be used as accelerators.

2.1 FPGA Programming Models

The software interface (i.e., programming model) for an FPGA is determined via a reserved portion of the FPGA called a *shell*, often provided by the manufacturer. The shell

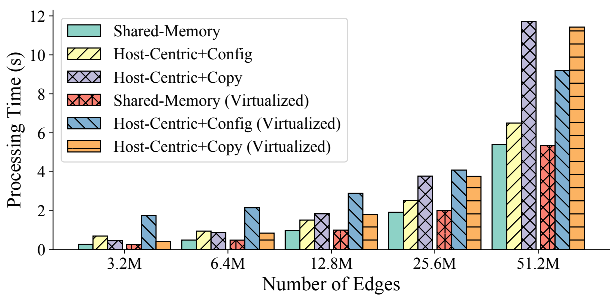


Figure 1. Graph processing time using the SSSP algorithm.

is responsible for sending, receiving, and processing I/O packets (such as those from the CPU, network, system memory, etc.), and generally presents one of two programming models to system software: *host-centric* or *shared-memory*. In both of these models, the shell exposes a memory-mapped IO (MMIO) control plane for software to manage the accelerator. The key difference between these models is whether accelerators can issue their own direct memory accesses (DMAs).

In the more widespread host-centric model, the accelerators are unaware of the system memory map and thus cannot issue DMAs. Instead, the CPU configures a DMA engine to transfer data from system memory to the accelerators. The host-centric model yields simpler hardware, as accelerator architects need not add DMA logic to their designs, instead relying on software programmers to manage DMAs.

However, the host-centric model incurs the latency of repeated communication between the CPU and accelerators for applications that exhibit pointer chasing. Specifically, the CPU must repeatedly configure the DMA engine to fetch new data for each accelerator. While scatter-gather DMA engines [81] can alleviate the penalty of certain non-contiguous access patterns (e.g., those where the sequence of DMA addresses is known prior to accelerator execution), they cannot alleviate the penalty of pointer chasing, as the sequence of DMA addresses is determined during accelerator execution.

In the emerging shared-memory model (e.g., that of Intel HARP [25]), each accelerator is cognizant of the system memory map and can issue its own DMAs. Therefore, shared-memory accelerators can engage in pointer chasing without interrupting the host to issue subsequent DMAs, avoiding the latency of host-centric platforms for such applications.

We use a graph processing application that uses the single source shortest path (SSSP) algorithm [89] to demonstrate the benefits of the shared-memory programming model. The algorithm needs to iteratively access a non-contiguous set of vertices and edges, thereby emulating the behavior of pointer chasing in the absence of scatter-gather DMA support (i.e., on our evaluation platform).

We implement this algorithm on Intel HARP, under the original shared-memory interface and a host-centric interface. Fig. 1 shows the processing time of the algorithm on a set of graphs with 800K vertices and an increasing number of edges. "Host-Centric+Config" indicates that the host-centric FPGA's DMA engine has been configured to fetch each individual data segment, while "Host-Centric+Copy" indicates that the host copies all data segments to a contiguous buffer before invoking the DMA engine. As shown, the shared-memory implementation is 17%–60% faster than that of the host-centric. The benefit of the shared-memory model is even more striking in a virtualized environment (37%–85% faster execution), where control plane operations become more expensive due to hypervisor trap-and-emulate. In sum, the DMA capabilities of shared-memory accelerators allow workloads to engage in pointer chasing without CPU involvement, reducing communication costs and improving performance.

2.2 FPGA Virtualization

The accelerators on an FPGA can be multiplexed spatially [15, 18, 37, 53, 55, 72, 74] and temporally [18, 37, 53, 55, 73, 84]. Spatial multiplexing allows different accelerator configurations to simultaneously occupy the same FPGA. Temporal multiplexing allows each individual accelerator configuration on an FPGA to be shared by multiple VMs. Temporal multiplexing can either be non-preemptive (i.e., run-to-completion) [73] or preemptive (i.e., pause-and-resume) [37].

To virtualize an FPGA, each virtual accelerator's on-FPGA resources as well as IO channels must be isolated [37]. The FPGA synthesizer handles most on-FPGA resource isolation. Specifically, the synthesizer ensures that each accelerator on a spatially-multiplexed FPGA is provisioned a distinct portion of device resources. If a physical accelerator is additionally overprovisioned via preemptive temporal multiplexing, accelerator designs must include support for saving and restoring their execution states upon preemption.

As for IO channels, FPGAs utilize both an MMIO control plane and a DMA data plane. Since software initiates all MMIO accesses in both the host-centric and shared-memory programming models, a hypervisor can easily virtualize guest access to MMIO registers via trap-and-emulate. In the host-centric model, software also initiates all DMAs, meaning host-centric DMAs can also be virtualized via trap-and-emulate [18] or paravirtualization [73].

However, in the shared-memory model, accelerators issue their own DMAs without software intervention, posing a problem for DMA virtualization. The traditional virtualization solution for DMA-capable IO devices has been a combination of SR-IOV [43] and PASID [28]. With SR-IOV, the IO memory management unit (IOMMU) provides a unique IO page table for each virtual device, thereby allowing the hypervisor to install unique address mappings that are enforced by the IOMMU at the time of DMA for each guest. With PASID, the IOMMU uses a CPU page table to translate DMAs, thereby allowing IO devices to directly access a process's address space. Each DMA is tagged with a process identifier, which the CPU uses to select the correct page table.

Unfortunately, the applicability of these techniques to shared-memory platforms is currently limited for two reasons. First, SR-IOV and PASID only virtualize PCIe links. Thus, on shared-memory platforms that expose both a UPI link and a PCIe link (e.g., Intel HARP [25]), SR-IOV and PASID cannot provide complete virtualization.

Second, the scalability of SR-IOV implementations in shared-memory FPGAs is severely limited. Although the SR-IOV standard supports thousands of VFs [43], shared-memory FPGAs have only supported one VF for the past five years [25, 27]. Because SR-IOV implementations are proprietary, our knowledge of the factors restricting scalability in shared-memory FPGAs is limited. However, certain shared-memory platforms such as Intel HARP [25] currently implement both the SR-IOV and the (related) IOMMU as soft IP in the FPGA shell, restricting scalability as compared to that of more resource-efficient hard IP implementations.

3 Goals and Challenges

OPTIMUS targets a use case in which cloud providers configure FPGAs as a set of popular accelerators for their customers, avoiding the penalty of virtual accelerator reconfiguration in favor of increased uptime [6, 18, 37]. To enable efficient and flexible sharing of accelerators on FPGAs, OPTIMUS utilizes spatial multiplexing [15, 18, 37, 53, 55, 72, 74] to partition an FPGA into a fixed set of accelerators, and temporal multiplexing [18, 37, 53, 55, 73, 84] to overprovision each of these accelerators. Because OPTIMUS novelly virtualizes shared-memory FPGAs, OPTIMUS tailors the goals of FPGA virtualization to shared-memory platforms as follows:

Programmability Unlike virtualization solutions for host-centric platforms [15, 18, 37, 53, 55, 72–74], OPTIMUS aims to share a unified virtual memory address space between software and hardware, similar to the original HARP interface [25]. However, programmability implies that cloud application developers should not have to deal with low-level platform details such as memory isolation, and should instead rely on straightforward memory abstractions of unified address spaces [1, 47, 58, 88]. Therefore, OPTIMUS must provide user-friendly abstractions for its unified CPU and FPGA address spaces to achieve programmability.

Isolation While host-centric FPGA virtualization solutions focus on the isolation of on-FPGA DRAM [15, 18, 37, 53, 72–74], OPTIMUS must consider the isolation of system memory in the presence of accelerator DMAs. Given limited support for hardware-assisted virtualization, OPTIMUS must provide strong DMA isolation within a single IOMMU address space.

We note that OPTIMUS assumes the synthesizer places each physical accelerator on isolated pieces of the FPGA fabric. Additionally, OPTIMUS does not consider side channels, which are an interesting direction for future work.

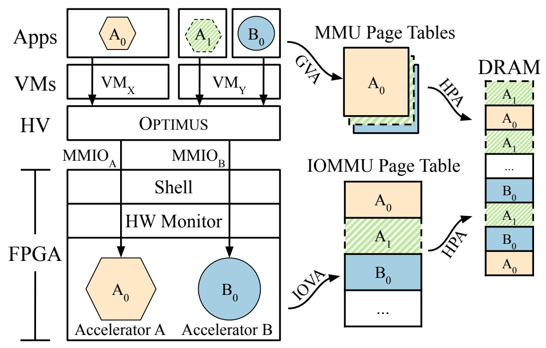


Figure 2. OPTIMUS design overview, shown with two physical accelerators for brevity. OPTIMUS spatially multiplexes a shared-memory FPGA as physical accelerators (A and B), and temporally multiplexes physical accelerators as virtual accelerators (A_0, A_1, and B_0).

Scalability As the number of accelerators on an FPGA increases, the FPGA's *multiplexers* (i.e., the hardware components that propagate signals between the set of accelerators and the singular system interconnect) must process data from a greater number of sources within timing constraints (e.g., a given number of cycles). At some point, a flat multiplexer arrangement physically cannot process all the signals under timing constraints; a multiplexer tree hierarchy must instead be used [37]. Given that OPTIMUS targets hardware operating at higher frequencies than state-of-the-art solutions—thereby placing tighter constraints on timing—OPTIMUS must provide a multiplexer tree by default to achieve scalability.

Efficiency OPTIMUS must have low virtualization overhead to provide sufficient performance to each VM. Specifically, the sum of each virtual accelerator's bandwidth must be as close as possible to the FPGA's total bandwidth. Furthermore, the latency added by hypervisor and hardware monitor execution must be minimized. Given the frequent occurrence of DMAs as compared to MMIOs, the primary challenge is ensuring that DMAs occur with minimal overhead. Unfortunately, traditionally-efficient DMA isolation methods such as SR-IOV and PASID do not currently provide a comprehensive and scalable DMA virtualization solution. Therefore, OPTIMUS must synthesize virtualization support into the FPGA to achieve the efficiency of hardware-assisted virtualization.

Fairness In line with prior work [37], OPTIMUS aims to ensure that each accelerator receives a fair share of the FPGA's total bandwidth. Given N spatially multiplexed physical accelerators, each accelerator must receive at least $1/N$ of the total real-time bandwidth when transmitting data. In temporal multiplexing, the physical accelerator must be assigned to each virtual accelerator for the same amount of time.

4 Design

Optimus follows a mediated pass-through [70] architecture in which control plane operations are trapped by the hypervisor, while data plane operations bypass the hypervisor. Fig. 2 shows the high-level architecture of Optimus, limited to two accelerators for brevity. Optimus uses the FPGA's shell to configure a shared-memory FPGA as a fixed set of *physical accelerators* (A and B), thereby offering spatial multiplexing. Optimus can additionally expand its virtualization scalability by temporally sharing a physical accelerator among several *virtual accelerators* (A_0 and A_1). For example, in Fig. 2, virtual accelerator A_0 is scheduled on physical accelerator A (meaning A holds A_0's execution state), while Optimus stores virtual accelerator A_1's execution state in DRAM until re-scheduling A_1 on physical accelerator A.

MMIO Control Plane Optimus traps all virtual accelerator control plane operations (MMIOs) to redirect the operations to the correct physical location. For scheduled virtual accelerators (A_0 and B_0), Optimus adds an offset to the trapped MMIOs in order to address the appropriate physical accelerator, forwarding the adjusted MMIOs to the FPGA. The hardware monitor then routes each MMIO to the appropriate physical accelerator (A or B) based on the offset MMIO address. For a queued virtual accelerator (A_1), Optimus postpones the MMIO access until the virtual accelerator is re-scheduled on a physical accelerator. The details of MMIO operations in temporal multiplexing will be discussed in §4.2.

DMA Data Plane Guest applications and their accelerators interact with DRAM using virtual addresses, which are translated to host physical addresses by the MMU and IOMMU respectively. However, the IO virtual addresses (IOVAs) used for virtual DMAs are offset versions of guest virtual addresses (GVAs). Although the CPU can provision a separate hardware page table in the MMU (i.e., an extended page table) for each application, only a single hardware page table is available to the FPGA in the IOMMU. Thus, Optimus must partition the single IO virtual address space among virtual accelerators using a technique called *page table slicing*, where each virtual accelerator's DMA region begins at a unique offset within the IO virtual address space. Optimus stores an offset table within the hardware monitor to translate from guest virtual addresses to IO virtual addresses during DMAs.

4.1 Hardware Monitor

Fig. 3 shows the FPGA configuration to support Optimus. The manufacturer provides the shell, which serves as the IO interface for the FPGA. Optimus uses the shell to load the cloud provider's desired accelerator configurations onto the FPGA. Optimus also includes a hardware monitor (shown in gray) on the FPGA.

Virtualization Control Unit Optimus uses the virtualization control unit (VCU) to configure the runtime behavior of

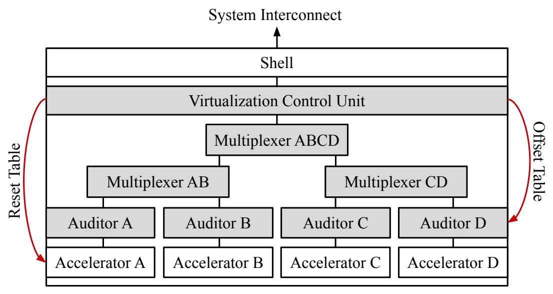


Figure 3. An example Optimus FPGA architecture, with the hardware monitor components shaded in gray. A two-level binary multiplexer tree is shown for brevity, but the multiplexer tree arrangement is configurable.

the hardware monitor. Specifically, VCU presents an accelerator management interface to allow Optimus to configure the offset and reset tables. The offset table stores offsets between guest virtual addresses and IO virtual addresses for each accelerator (necessary to support page table slicing). The reset table is used to specify the reset signal for each accelerator, thus enabling Optimus to reset individual accelerators to clear state for isolation purposes on a VM context switch.

Optimus reserves a special region of MMIO for communication with VCU. If the incoming packets fall in this range, the virtual control unit intercepts the packets to configure the hardware monitor. Otherwise, VCU forwards the packets to the multiplexer tree.

Multiplexer Tree The multiplexer tree is responsible for propagating input packets from the shell to each accelerator, and transmitting output packets from each accelerator to the shell. Each multiplexer in the multiplexer tree operates on a round robin scheduling policy, thereby ensuring equal bandwidth for each accelerator on the same path through the multiplexer tree (and thus, fair real-time bandwidth sharing as mentioned in §3). However, if cloud providers seek to provide greater bandwidth to some accelerator A, the multiplexer tree can be configured to place fewer accelerators under the multiplexers on A's path.

Auditors Unlike AXI or Avalon interconnects [31, 79], the multiplexer tree does not make routing decisions based on the accessed address. Instead, the multiplexer tree propagates packets to a set of *auditors* (one per physical accelerator), where each auditor determines whether incoming packets are intended for its associated accelerator. This lazy packet routing (i.e., waiting until the packets arrive at the auditor to make routing decisions) results in simpler circuitry than eager packet routing (i.e., including routing logic within the multiplexer tree).

If the incoming packet is an MMIO, the auditor checks that the MMIO offset falls within the accelerator's MMIO range. If so, the auditor forwards the packet to its associated accelerator. If not, the packet is discarded.

If the incoming packet contains DMA data, the auditor must determine if the packet is a response to a DMA that the accelerator initiated. When an accelerator wishes to perform a DMA, the auditor tags the outgoing packet with an accelerator ID, which is preserved in the response packet. Thus, an auditor can verify if a DMA packet is intended for its accelerator by checking the packet's accelerator ID field. If so, the packet is forwarded to the accelerator. If not, the packet is discarded.

Page Table Slicing For simplicity, guest applications and their virtual accelerators would both access memory using guest virtual addresses, which would ultimately be translated to host physical addresses by the MMU and IOMMU respectively. However, given the limitation of a single IO virtual address space, the guest virtual addresses of different applications would conflict if used as keys in the IO page table. To isolate guest memory, OPTIMUS introduces a hardware-software co-design called page table slicing, which adapts prior software-only techniques for virtualizing GPUs [70] and wireless NICs [78].

Page table slicing configures the auditors with a *linear* address mapping policy, where guest virtual addresses (GVAs) map to IO virtual addresses (IOVAs). OPTIMUS allows each accelerator to access a contiguous *DMA memory* range $[g, g+p)$ in the application's address space. It also divides IOVAs into several p-sized partitions, and assigns each partition to a unique (virtual) accelerator. For a given IOVA partition $[i, i+p)$, OPTIMUS stores the offset value $(i-g)$ in the corresponding accelerator's entry in the offset table. Afterward, the accelerator's auditor can convert between IOVAs and GVAs during DMAs within a single cycle, ensuring efficient memory isolation. In the presence of temporal multiplexing (i.e., oversubscription of individual accelerators), OPTIMUS updates the physical accelerator's offset table entry with the newly-scheduled virtual accelerator's offset entry.

We consider page table slicing as a lightweight isolation method which is complementary to SR-IOV. Specifically, even if SR-IOV scalability increases for future shared-memory FPGAs, page table slicing would allow for nested virtualization on SR-IOV enabled devices; a cloud provider could use SR-IOV to provide a "vFPGA" to a VM acting as a nested hypervisor. The nested hypervisor could then use page table slicing to share this vFPGA among its own guests.

Shadow Paging An important goal of OPTIMUS is to share a contiguous range of virtual memory between software and hardware, which requires the IOMMU (together with page table slicing) to directly map GVAs to HPAs. Since the IOMMU does not support nested paging, OPTIMUS maintains a shadow page table for each accelerator.

4.2 Preemption Interface

While spatial multiplexing allows different accelerators to run on the same FPGA, OPTIMUS uses temporal multiplexing to share a fixed accelerator configuration among different VMs, with each VM's virtual accelerator occupying the physical accelerator for a short time-slice. OPTIMUS must be able to preempt acceleration jobs to provide fair temporal multiplexing, and therefore exposes a *preemption interface* similar to that of AmorphOS [37].

A preemption-capable accelerator should implement a set of *control registers* which serves two purposes: 1) saving and restoring internal execution states, and 2) starting, preempting, and resuming acceleration jobs. Control registers are privileged resources, thus should not be accessible by virtual machines directly. The hypervisor traps and emulates accesses to control registers, and hides the hardware status of the physical accelerator. Registers besides the control registers are called *application registers*. Accesses to application registers are postponed until the virtual accelerator is scheduled. Specifically, if the register does not have side effects (i.e., read/write to the register is idempotent), the hypervisor can cache the register's value in software and synchronize the cache and the physical register while scheduling.

During virtual accelerator initialization, the accelerator informs OPTIMUS how much memory is needed to store internal execution states. OPTIMUS then allocates a memory buffer for the states and informs the physical accelerator of the buffer's base address via the control registers.

When OPTIMUS wishes to schedule a virtual accelerator on a physical accelerator, OPTIMUS reads the current job status from the physical accelerator. If the physical accelerator is occupied, OPTIMUS sends a preempt command, causing the physical accelerator to write the virtual accelerator's execution state to the system memory buffer. Once all in-flight transactions have been processed, the accelerator notifies OPTIMUS that context has been successfully saved and a new job may be scheduled, as in prior work [37]. If an accelerator fails to cede control, OPTIMUS can forcibly reset the accelerator after a configurable timeout period.

Later, when OPTIMUS re-schedules the original virtual accelerator job on the physical accelerator, it issues a resume command that instructs the physical accelerator to load execution state from its memory buffer and continue execution.

OPTIMUS's decision to leave the implementation of preemption to accelerator designers is a complexity-performance trade-off. On one hand, designers using OPTIMUS must reason about the state to save upon preemption, in contrast to automatic mechanisms such as Cascade [59]. On the other hand, designers using OPTIMUS can identify the minimal amount of state to save. For example, when preempting a linked-list walker, saving the address of the next node can be sufficient. In contrast, Cascade conservatively requires all latches to be saved. This results in a more complex circuit, consuming more resources, inhibiting a circuit's ability to scale to higher frequencies, and ultimately hurting performance. Thus, given OPTIMUS's performance and scalability goals, OPTIMUS relies on accelerator designers to implement the preemption interface.

4.3 Userspace API

Because native platform APIs can be complex [29], OPTIMUS offers a simplified API for software application developers. OPTIMUS provides a separate implementation of the same simplified API to accelerator developers for use in Verilog simulations.

From the guest's perspective, each accelerator is a PCIe device. OPTIMUS offers a customized driver and a userspace library that work in tandem to allow for application-level programming of accelerators. The driver is responsible for initializing the virtual accelerator, including mapping MMIO regions to userspace and registering DMA memory with the hypervisor. The userspace library allows the programmer to easily connect to and disconnect from a virtual accelerator, reset the accelerator, program the virtual accelerator through its MMIO region, and manage DMA memory.

5 Implementation

OPTIMUS is implemented atop the Intel HARP shared-memory FPGA platform [25] using Intel's Core Cache Interface (CCI-P) [24]; however, OPTIMUS's design can be applied to any shared-memory FPGA platform with IOMMU support (which is necessary to implement page table slicing). OPTIMUS is implemented as a kernel module in 3,199 lines of C code, using the *vfio-mdev* [36] framework for device mediation and KVM [39] for CPU and memory virtualization. The guest FPGA driver and user API library are an additional 2,033 lines of C code, not including a ported memory allocation library [46] used to help manage DMA regions for accelerators. The Verilog implementation of the hardware monitor relies on Intel's open-source multiplexer (MUX) module [33], which adds 1,237 lines of code. Altogether, the hardware monitor occupies less than 7% of on-FPGA configurable resources.

FPGA Interface HARP's shell provides a request/response interface called CCI-P for memory access [24], which encapsulates PCIe and UPI transactions. In order to access CPU memory, an accelerator sends a request packet and then waits for a corresponding response packet. While waiting, the accelerator may send out other requests to saturate the bandwidth.

MMIO Slicing The MMIO address space of OPTIMUS consists of three portions. The first portion of the MMIO space is reserved for the HARP shell. The next 4 KB is reserved for the virtualization control unit's accelerator management interface, via which the hypervisor can configure the hardware monitor (e.g., the offset and reset tables) and obtain the FPGA configuration information (e.g., the number of physical accelerators on the device and whether or not the configuration is compatible with OPTIMUS). Finally, each physical accelerator receives a 4 KB page for its individual MMIO state, with isolation enforced by the accelerator's auditor.

Guest-MMIO Layout From a guest's perspective, a virtual accelerator is a PCIe device. PCIe BAR0 points to the accelerator MMIO space, and PCIe BAR2 points to the hypervisor MMIO space (used to communicate with the hypervisor).

Page Table Slicing By default, OPTIMUS uses a 64 GB slice of the 48-bit IO virtual address space for each virtual accelerator. However, this can be increased on systems where more than 64 GB of RAM is needed per virtual accelerator.

OPTIMUS's guest library uses the `mmap()` system call with the `MAP_NORESERVE` flag to reserve a 64 GB slice without allocating physical memory or swap. OPTIMUS writes the base address of each slice to a register in BAR2 (the hypervisor MMIO space). The slicing offset is calculated based on the value stored in this register.

Shadow Paging For prototype simplicity, OPTIMUS currently features a hypercall-style shadow paging mechanism, reserving a register in the hypervisor MMIO space. During the initialization of each accelerator, OPTIMUS allocates a 2 MB page, and initializes the IOPT entries of the accelerator to map to the physical address of the page. When a guest wants to make a page FPGA-accessible, it uses this register to notify the hypervisor of the GVA and GPA for the page. The hypervisor then checks page permissions, calculates the correct IOVA and HPA, pins the HPA in memory, and inserts the IOVA$\rightarrow$HPA mapping into the IO page table.

Multiplexer Tree Hierarchy OPTIMUS uses a three-level binary tree which supports up to 8 physical accelerators. We experimented with different hierarchies for the multiplexer tree (e.g., more layers and more nodes per layer); however, for some benchmarks, the synthesizer was unable to synthesize greater than eight accelerator instances on the FPGA without lowering the multiplexer tree frequency below 400 MHz, which is necessary to fully utilize the memory bandwidth. Hence, we limited the tree's support to eight physical accelerators.

AMORPHOS [37]—a prior FPGA virtualization solution—uses a flat multiplexer to avoid the complexity and latency of a multiplexer-tree when there are eight or fewer accelerators, and uses a layered multiplexer-tree when there are greater than eight accelerators. However, in OPTIMUS, a flat multiplexer is not feasible even with a smaller number of accelerators, as it prevents OPTIMUS from multiplexing the accelerators at a high frequency (400 MHz).

Huge Pages In line with prior work [2, 4, 5, 11, 44, 45, 49], OPTIMUS uses huge pages to avoid IOTLB (IO translation lookaside buffer) thrashing and improve DMA performance. To the best of our knowledge, on the Intel HARP platform, the IOTLB for both 4 KB pages and 2 MB pages can only store 512 IOVA to HPA mappings. Only using 4 KB pages may cause frequent IOTLB misses, which hurts performance on HARP. OPTIMUS uses 2 MB huge pages for DMA memory, thereby allowing the IOTLB to cache 2 MB $*$ 512 = 1 GB worth of mappings.

We do not see 2 MB pages as a significant drawback for three reasons. First, hypervisors are already unable to oversubscribe memory in the presence of pass-through or SR-IOV-enabled devices; the device-accessible memory pages must be pinned due to the IOMMU's inability to handle page faults. Second, as opposed to pass-through or SR-IOV, OPTIMUS only pins FPGA-accessible pages once they are allocated by the guest. Third, data center servers are often equipped with hundreds of gigabytes of memory; therefore, 2 MB pages are relatively small.

IOTLB Conflict Mitigation When using our original page table slicing technique (in which each 64 GB slice is laid out contiguously in the IO virtual address space), we discovered that IOTLB mappings for different virtual accelerators were frequently evicting each other, hurting system performance.

While the exact eviction policy for the IOTLB is unknown, we believe the problem stems from a conflict in the set indices of IOVAs for different virtual accelerators. To the best of our knowledge, when the page size is 2 MB, the IOTLB uses 9 bits after the 21-bit huge page offset as the set index (bits 21-29). We believe each set consists of a single entry. Thus, if a virtual accelerator accesses a virtual page with the same set index as another virtual accelerator's page, an IOTLB conflict will occur. More precisely, a given page p_1 will conflict with any page p_2 where $p_1 \equiv p_2 \mod 2^9$.

To work around this problem in software (given the IOTLB could not be altered), we added an extra 128 MB of address space between each 64 GB IOVA slice to offset the set indices of different virtual accelerator pages. Because OPTIMUS supports eight physical accelerators and the IOTLB can address 1 GB of memory without conflicts, OPTIMUS divides this 1 GB of memory evenly among the accelerators, yielding 128 MB per accelerator. Thus, each virtual accelerator's working set must exceed 128 MB before IOTLB conflicts potentially occur among accelerators. If sequential accesses are performed, IOTLB misses are rare, regardless of the working set size.

Tiling and Partial Reconfiguration Like other FPGAs [6, 16, 56], HARP FPGAs can be reconfigured at tile granularity (i.e., a manufacturer-defined portion of the fabric). The reconfiguration of an individual tile is known as partial reconfiguration. However, HARP only provides a single tile, and therefore would require re-flashing all spatially-multiplexed accelerators to reconfigure an individual accelerator. As such, OPTIMUS does not support partial reconfiguration.

Temporal Multiplexing Interface For flexible memory management, each guest application allocates a buffer in host DRAM for storing accelerator state upon preemption.

Time Slice in Temporal Multiplexing The time slice used for temporal multiplexing is configurable; however the default value is 10 ms. A 10 ms time slice is possible because OPTIMUS does not reconfigure the FPGA upon preemption, since the temporally-multiplexed accelerators on a given physical accelerator share the same configuration. If partial reconfiguration support is added in the future, the time slice would need to be increased to allow for sufficient time to reconfigure individual tiles.

Temporal Multiplexing Scheduling OPTIMUS uses unweighted round-robin (i.e., equal time slices) as the default scheduling algorithm. However, OPTIMUS also implements a scheduler with weighted time slices and a priority-based scheduler.

6 Evaluation

In this section, we evaluate our prototype implementation of OPTIMUS and answer the following questions:

Efficiency What is the overhead of the hardware monitor in terms of FPGA resource utilization? To what extent does spatial multiplexing improve FPGA resource utilization (§6.2)? How much virtualization overhead does OPTIMUS incur compared to pass-through (i.e., direct assignment) (§6.3)? How does the use of huge pages influence memory throughput and latency? (§6.5)

Scalability How does OPTIMUS scale with respect to the number of acceleration jobs concurrently executing on the FPGA (§6.4)? How does OPTIMUS scale with respect to the oversubscription factor of each accelerator (i.e., the number of virtual accelerators per physical accelerator) (§6.6)?

Fairness How similar is the DMA bandwidth for each physical accelerator (§6.7)? Does OPTIMUS enforce different scheduling policies among its virtual accelerators (§6.8)?

6.1 Experimental Setup

Hardware We evaluate OPTIMUS on Intel Skylake HARP [25]. The platform features a 2.8 GHz Xeon CPU and a 400 MHz Arria 10 FPGA [32] located in the same package. The CPU and FPGA are connected via a single UPI [51] link as well as two PCIe 3.0 links. The server has 188 GB of DRAM.

Software OPTIMUS runs CentOS 7.5 with Linux kernel version 5.1.0-rc6 as the host OS, using QEMU version 3.0.1. Each guest also runs CentOS 7.5 and is allocated 10 GB of the server's 188 GB of DRAM.

Baseline We compare OPTIMUS's performance with virtualization via pass-through (i.e., direct assignment). To allow the FPGA to directly access the application's virtual address space, we enable vIOMMU [83] (virtual IOMMU) support in QEMU. To our knowledge, there are no shared-memory FPGA hypervisors to which we can compare OPTIMUS.

Configuration Unless mentioned specifically, OPTIMUS uses 2MB huge pages with IOTLB Conflict Mitigation enabled.

App	Description	LoC	Freq. (MHz)
AES	AES128 Encryption Algorithm	1965	200
MD5	MD5 Hashing Algorithm	1266	100
SHA	SHA512 Hashing Algorithm	2218	200
FIR	Finite Impulse Response Filter	1090	200
GRN	Gaussian Random Number Generator	1238	200
RSD	Reed Solomon Decoder	5324	200
SW	Smith Waterman Algorithm	1265	100
GAU	Gaussian Image Filter	2406	200
GRS	Grayscale Image Filter	2266	200
SBL	Sobel Image Filter	2451	200
SSSP	Single Source Shortest Path	3140	200
BTC	Bitcoin Miner	1009	100
MB	Random Memory Accesses	1020	400
LL	Linked List Walker	695	400

Table 1. The benchmarks used to evaluate OPTIMUS, the number of lines of Verilog code used to implement benchmarks, and the frequencies at which benchmarks are executed.

Benchmarks Table 1 shows the fourteen benchmarks with which we evaluate OPTIMUS. Ten of these benchmarks are ported from HardCloud [17], an open-source framework that offloads OpenMP [20] computation tasks to the FPGA. Our HardCloud benchmarks are all compute-intensive; they include signal processing, cryptography, scientific computing, and image processing applications. We port these benchmarks to our virtualization platform, and use their default configuration during synthesis. Besides, we also port an FPGA based graph processing application (single source shortest path or SSSP) [89], and a bitcoin miner [3] to our virtualization platform. Unlike in §2.1, we only evaluate the shared-memory implementation of SSSP in this section, while configuring the benchmark to use a graph with 800K vertices and 12.8M edges. HardCloud benchmarks, SSSP, and Bitcoin are chosen to represent real-world applications.

Since no open-source benchmarks for HARP place sufficient strain on OPTIMUS's bandwidth and latency for a single acceleration job, and because no existing benchmarks conform to OPTIMUS's preemption interface, we provide two benchmarks ourselves. Both of these benchmarks implement OPTIMUS's preemption interface in order to evaluate OPTIMUS's temporal multiplexing capabilities.

MemBench (MB) *concurrently* issues random DMA read and write requests in order to saturate HARP's bandwidth. The random reads and writes result in the worst-case effects of IOTLB misses, and thus minimize throughput benefits from memory locality.

LinkedList (LL) *sequentially* fetches cache line sized nodes from a linked list distributed randomly in DRAM, connecting the performance of LinkedList to worst-case DMA patterns and thus creating a latency bottleneck. Because shared-memory FPGAs are an emerging technology, there are currently few open-source benchmarks that leverage this model. However, LinkedList represents the fundamental limitations for irregular parallel applications (i.e., with a lot of pointer chasing), and prior work [77] has demonstrated that linked lists are sufficient to study the overhead of latency-bound workloads on shared-memory FPGA platforms.

The latency sensitivity of LinkedList requires special treatment due to the intricacies of the HARP platform. All HardCloud benchmarks allow the HARP shell to automatically select the interconnect channel (PCIe or UPI) used for each IO packet; for throughput-bound workloads, this configuration generally yields optimal performance [24]. However, for a highly latency-sensitive benchmark such as LinkedList, automatic channel selection yields unstable performance. HARP's channel selector is optimized for throughput rather than latency. Thus, although UPI has lower latency for reads [24], the channel selector places some reads on PCIe, leading to wide performance variation for latency-sensitive benchmarks. As such, we measure the performance of LinkedList under two configurations: PCIe-only and UPI-only.

Table 1 shows the frequency at which each benchmark is run. Ideally, each benchmark would be run at the highest frequency that the FPGA board supports (400 MHz). However, a number of the benchmarks are too complex for HARP's current synthesizer to be able to ensure that their circuits can correctly operate at this maximum frequency; the synthesizer cannot place the FPGA logic elements sufficiently close in order to propagate signals quickly enough. We therefore synthesize each benchmark at the highest frequency achievable with OPTIMUS's maximum number of physical accelerators (eight). As synthesis algorithms improve, we anticipate being able to run the benchmarks at higher frequencies.

6.2 FPGA Resource Utilization

In this section, we evaluate the impact of OPTIMUS on FPGA resource utilization as reported by Intel's FPGA toolchain. We measure the percent of on-FPGA resources consumed by the hardware monitor (indicating virtualization overhead), and we explore the extent to which spatial multiplexing can improve FPGA resource utilization.

Table 2 displays the percentage of Adaptive Logic Modules (ALMs) and Block RAM (BRAM) that each major FPGA component utilizes on (1) a single accelerator pass-through baseline versus (2) eight accelerators under OPTIMUS. The FPGA shell is an inherent component in both OPTIMUS and the pass-through baseline, and consumes 23.44% of ALMs and 6.57% of BRAM. The hardware monitor is only present in OPTIMUS, but utilizes just 6.16% of the ALMs and 0.48% of the BRAM, indicating low virtualization overhead in terms of resource utilization.

Without the spatial multiplexing of OPTIMUS, benchmarks in the pass-through accelerator configuration utilize no more than 5% of available FPGA resources. OPTIMUS's spatial multiplexing increases aggregate accelerator resource utilization roughly linearly. With eight accelerators, OPTIMUS's

FPGA Component		ALM Usage (%)		BRAM Usage (%)	
		OPTIMUS	PT	OPTIMUS	PT
Shell		23.44	23.44	6.57	6.57
Hardware Monitor		6.16	0.00	0.48	0.00
App	AES	27.80	3.62	23.01	2.82
	MD5	34.27	4.35	23.01	2.82
	SHA	18.16	2.16	22.46	2.82
	FIR	15.77	1.92	22.46	2.82
	GRN	12.53	1.76	7.98	1.02
	RSD	17.93	2.21	22.87	2.87
	SW	10.34	1.42	11.67	1.47
	GRS	9.92	1.32	18.15	2.28
	GAU	25.28	3.41	21.24	2.60
	SBL	18.49	2.39	20.30	2.55
	SSSP	15.73	1.96	22.47	2.82
	BTC	8.99	1.32	4.16	0.48
	MB	4.84	0.83	0.00	0.00
	LL	-0.24	0.15	0.00	0.00

Table 2. Breakdown of FPGA resource utilization by component (ALM and BRAM). Each component's utilization is reported as a percentage of the total amount of each resource type available on the FPGA. The pass-through (PT) baseline features a single instance of the accelerator benchmark, while OPTIMUS features eight instances in order to compare resource utilization in the presence of spatial multiplexing.

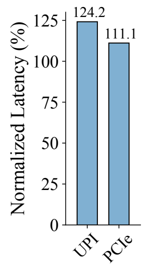
slight overhead beyond 8x stems from increased circuit complexity as the number of accelerators increases. Specifically, the synthesizer must consume extra resources in order to route signals to different locations on the FPGA chip under timing requirements.

MemBench and LinkedList are sufficiently simple that the synthesizer is able to optimize the FPGA configuration, yielding a sublinear relationship. MemBench only uses 6x the number of ALMs as the pass-through baseline. As for LinkedList, overall resource usage actually decreases, and is thus listed as using a negative portion of resources in Table 2.

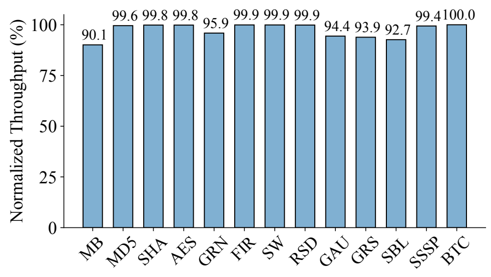
6.3 Performance Overhead

To measure the virtualization overhead introduced by OPTIMUS, we compare the performance of an accelerator virtualized via pass-through (i.e., direct assignment) with an accelerator virtualized via OPTIMUS, as shown in Fig. 4.

Latency Fig. 4a shows the latency overhead for LinkedList—a microbenchmark which represents the worst-case for latency-bound applications—when running in PCIe-only mode and UPI-only mode. The 24% latency overhead of LinkedList stems from a decision to favor scalability over latency in the arrangement of our hardware multiplexers. In order to pass timing requirements when scaling to eight accelerators, we require a three-level binary tree (as opposed to a single multiplexer with eight child accelerators). Unfortunately, each added layer of the tree adds approximately 33 ns of latency; therefore, our design induces approximately 100 ns of latency



(a) Latency



(b) Throughput

Figure 4. Performance overhead of different benchmarks compared to pass-through.

on the path through the multiplexer tree in order to provide scalability.

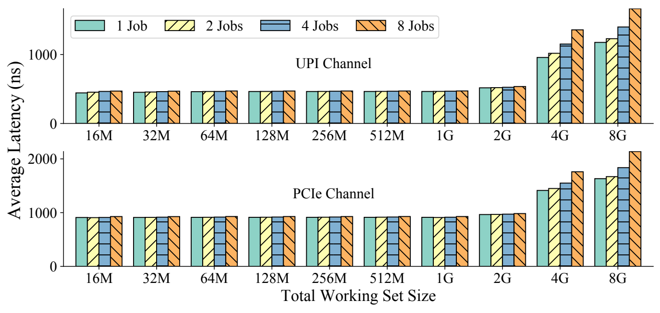
Throughput Fig. 4b displays the throughput overhead for the remaining benchmarks. For MemBench (a microbenchmark which represents the worst-case for bandwidth-intensive applications), the relative throughput overhead is 9.9%. MemBench is specifically designed to stress the interconnection as much as possible, and therefore issues memory requests at every possible FPGA cycle. However, given the routing complexity of the multiplexer tree, the accelerator can only transmit a memory request packet every two cycles. Thus, the multiplexer tree is again the primary source of overhead. Despite this worst-case scenario, our HardCloud benchmark results indicate that the throughput overhead of OPTIMUS is less than 5% for realistic applications.

6.4 Scalability of Spatial Multiplexing

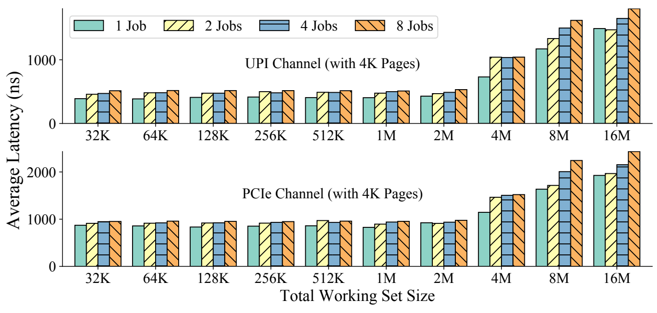
In this section, we assess OPTIMUS's ability to scale with respect to the number of acceleration jobs executing concurrently on the FPGA. For each benchmark, we place eight instances of the accelerator on the FPGA (i.e., the maximum number of physical accelerators that can be synthesized on our platform). We measure the performance of each benchmark as the number of concurrent acceleration jobs increases.

Latency Because LinkedList is highly sensitive to memory access latency, we measure the benchmark's execution time as the number of acceleration jobs increases to determine the effects of scaling on latency. As shown in Fig. 5a, increasing the number of acceleration jobs has negligible effect on aggregate latency if the working set does not exceed IOTLB capacity. The slight ($< 6\%$) increase from 1 job to 8 jobs is due to IO queuing delays.

When the working set barely exceeds IOTLB capacity (2G), latency only suffers a slight increase, since address translation is not overwhelmed. However, once the working set reaches 4G, the queuing delay is exacerbated by frequent address

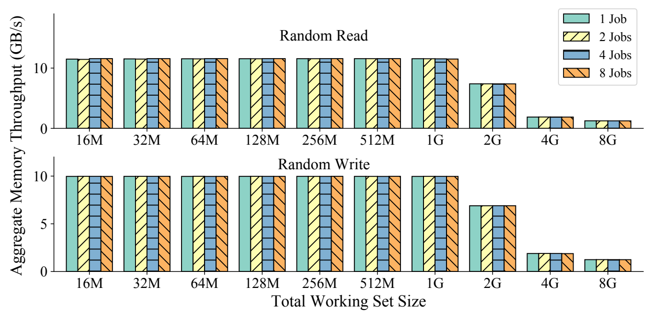


(a) With 2M Pages

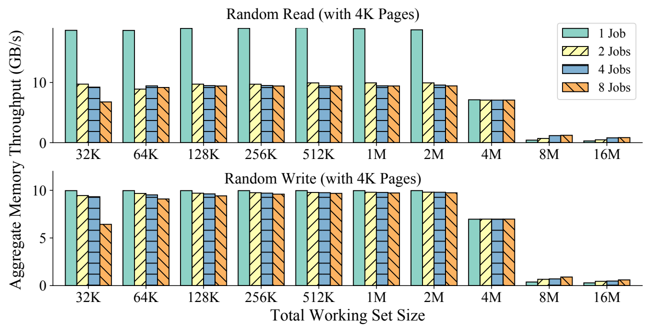


(b) With 4K Pages

Figure 5. Average memory access latency of LinkedList with different working set sizes and number of virtual machines.



(a) With 2M Pages



(b) With 4K Pages

Figure 6. Aggregate throughput of MemBench with different working set sizes and number of virtual machines.

translation, resulting in a rapid increase in average latency as the number of jobs grows.

Throughput Since a single instance of MemBench saturates the platform's bandwidth, MemBench indicates the worst-case measurement of throughput scalability. Fig. 6a shows the aggregate throughput of MemBench as the number of acceleration jobs and aggregate working set size are increased. As demonstrated, increasing the number of acceleration jobs does not diminish the aggregate throughput. Thus, OPTIMUS scales well in terms of memory access throughput.

The drop-off in throughput beyond 1 GB is not due to OPTIMUS, but rather due to the limitations of the current HARP IOMMU. Since we believe that the IOTLB only contains 512 entries when the page size is 2 MB, the IOTLB is limited to only caching the mappings of 1 GB virtual address space. Thus, when the aggregate working set size exceeds 1 GB, throughput degrades as a result of IOTLB misses.

In HARP, the IOMMU is not integrated into the CPU in order to minimize CPU modifications needed to support the experimental platform. As a result, upon each IOTLB miss, the IOMMU must go through the system interconnection to fetch the required IO page table from the CPU. We argue that in future generations of shared-memory FPGA platforms, the manufacturer should increase the number of IOTLB entries and integrate the IOMMU into the CPU in order to mitigate the frequency and severity of IOTLB misses. Additionally, supplementing a CPU-integrated IOMMU with hard-wired support for SR-IOV could potentially allow SR-IOV to scale on shared-memory platforms. Further modifying the IOMMU to support SR-IOV on UPI links could even allow SR-IOV to virtualize encapsulated PCIe and UPI transactions.

Fig. 7 shows the aggregate throughput (normalized to a single acceleration job) of our real-world applications as the number of acceleration jobs is increased. Unlike MemBench, none of these applications fully utilize the bandwidth for a single acceleration job. As a result, the aggregate throughput increases as the number of acceleration jobs increases. Except for Gaussian, Grayscale, Sobel, and Bitcoin, whose working set sizes are relatively small, the total working set sizes of other applications vary from 2GB to 32GB, which means the capacity of IOTLB is exceeded. However, since all these applications are well-designed to have good memory locality, performance is not impacted due to IOTLB thrashing.

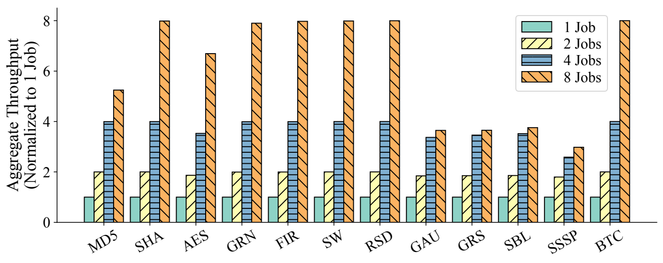


Figure 7. The aggregate throughput of different real-world applications, normalized to the throughput of a single VM. GAU, GRS, SBL, and SSSP fail to scale because the interconnection bandwidth becomes saturated, creating a performance bottleneck beyond four accelerators.

6.5 Benefit of Using Huge Pages

To measure the performance benefit of "huge" (2M) pages, we compare the throughput and latency when using 2M versus 4K pages. Fig. 5 and Fig. 6 compare the results of 2M versus 4K paging in terms of latency and throughput, respectively.

Optimus suffers from a performance drop when the aggregate working set exceeds the IOTLB capacity (512 pages); a 2M TLB entry can serve 512 times more memory than a 4K entry. Using 2M pages can thus postpone the performance drop from a 4M aggregate working set to 2G, which is beneficial for applications with a large working set.

As shown in Fig. 6b, we discovered an unusually-high read throughput when (a) there is only one accelerator, and (b) the working set does not exceed 2M. We noticed a similar phenomenon with 2M pages, which is not pictured due to spacing constraints. While we cannot definitively determine the source of this behavior, we believe the phenomena arise due to a speculative optimization in the IOTLB pipeline, which assumes that subsequent memory accesses will access the same 2 MB region as previous accesses.

6.6 Scalability of Temporal Multiplexing

In this section, we evaluate how Optimus scales with respect to the oversubscription factor (i.e., the number of virtual accelerators per physical accelerator). Since only MemBench and LinkedList conform to Optimus's preemption interface, we are limited to directly evaluate these benchmarks. However, preemption overhead is correlated to the amount of execution state that must be saved. Therefore, because we know the total set of resources consumed by each accelerator configuration, we can use this percentage as an upper bound on the amount of state that must be saved, thus establishing an upper bound on context-switching overhead.

Fig. 8 presents the aggregate throughput of running a varying number of virtual accelerators on a physical accelerator, normalized against a single job on an accelerator. Theoretically, Optimus does not have a hard limitation on the scalability of temporal multiplexing. Our evaluation stops at 16

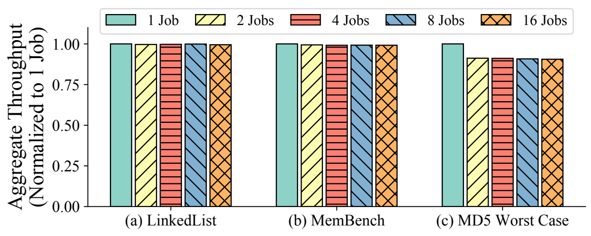


Figure 8. Normalized aggregate throughput in the presence of preemptive temporal multiplexing. All virtual accelerators are scheduled on a single physical accelerator.

because we are able to show that the context-switching overhead does not increase as the number of jobs increases.

As indicated by the drop-in throughput between 1 and 2 jobs, the overhead of preemption for LinkedList is approximately 0.5%. For MemBench, this number is 0.7%. The overhead remains constant beyond 2 jobs because preemption occurs at a fixed interval in the presence of temporal multiplexing, regardless of the number of jobs being multiplexed.

We estimate the worst-case overhead of temporal multiplexing for real-world applications by simulation. Since MD5 occupies the most on-FPGA resources of any real-world application, we use this benchmark to establish an upper bound. Our estimation yields 9% temporal multiplexing overhead in the worst case (i.e., assuming all resources occupied by MD5 must be saved on a context switch).

We stress that the amount of state that must be saved is application-dependent. If the amount of state is large, the length of each time slice can be increased to reduce the number of context switches, thereby mitigating the penalty.

6.7 Fairness of Spatial Multiplexing

In this section, we measure the fairness of the hardware scheduler in terms of its ability to guarantee at least $1/N$ of the total real-time bandwidth to each of N physical accelerators, assuming those accelerators are actively transmitting data. We assess the bandwidth fairness in both homogeneous configurations (where the FPGA is configured with multiple instances of the same accelerator) and heterogeneous configurations (where the FPGA is configured with various accelerators).

Homogeneous Configurations For each benchmark, we configure the FPGA with eight homogeneous accelerators and measure the per-accelerator throughput. Table 3 presents the *normalized throughput range* (i.e., the difference between the maximum and minimum accelerator throughput divided by the average throughput) for each benchmark. The maximum normalized throughput range is approximately 1%, demonstrating that the difference in throughput between any two accelerators is at most 1%. In other words, given eight homogeneous accelerators, each accelerator achieves roughly 1/8 of the aggregate throughput. Thus, the hardware monitor

Accelerators	AES	MD5	SHA	FIR	GRN	RSD	SW	GAU	GRS	SBL	SSSP	BTC	MB	LL
Normalized Throughput Range (10^{-4})	21.9	11.9	4.40	30.1	108	1.77	3.79	63.1	1.60	147	595	0.468	1.83	3.25

Table 3. Normalized throughput range among eight homogeneous physical accelerators.

Co-located Accelerator	AES	MD5	SHA	FIR	GRN	RSD	SW	GAU	GRS	SBL	SSSP	BTC	MB	LL
Normalized Throughput	0.86x	0.50x	0.77x	0.75x	1.00x	0.78x	0.78x	0.80x	0.80x	0.79x	0.75x	1.00x	0.50x	1.00x

Table 4. MemBench's throughput when co-located with a second active accelerator, normalized against a standalone instance.

fairly multiplexes the FPGA among physical accelerators in homogeneous FPGA configurations.

Heterogeneous Configurations MemBench is designed to saturate HARP's bandwidth for a single job. Therefore, we use it as a baseline for full throughput, and measure the relative decrease in MemBench's throughput in the presence of a second active accelerator benchmark.

Table 4 shows the normalized throughput reported by the MemBench accelerator for each configuration. In the presence of a second active accelerator, MemBench is guaranteed to receive at least half of the original bandwidth.

Upon first glance, MemBench receiving more than half of the total bandwidth may appear to be unfair. However, most accelerators do not transmit data as often as MemBench. For instance, in the cases where data is rarely transmitted by the other accelerator (e.g., LinkedList), MemBench receives a near-complete share of the bandwidth. When the second accelerator is also bandwidth-hungry (e.g., MD5 and a second instance of MemBench), the bandwidth is evenly split.

6.8 Fairness of Temporal Multiplexing

Enforcing fairness in the context of a software scheduler means being able to enforce the cloud provider's custom time-sharing policy. OPTIMUS implements an unweighted round-robin scheduler (i.e., equal time slices), a weighted scheduling policy (i.e., weighted time slices), and a priority scheduler (i.e., the job with the greatest priority runs at each time slice). We verify that the software scheduler successfully enforces each policy by measuring the execution time of each virtual accelerator across varying oversubscription factors, time slice lengths, and job weights/priorities. On average, the actual execution times are within 0.32% of the expected times, with the greatest difference being 1.42%. Thus, OPTIMUS successfully enforces each of its software scheduling policies.

7 Discussion

7.1 OPTIMUS vs. AMORPHOS

AMORPHOS [37] targets OS management of FPGAs. Like OPTIMUS, AMORPHOS enables both spatial and temporal multiplexing of FPGAs. AMORPHOS overcomes the static limitations of partial reconfiguration (i.e., forcing accelerator designs to fit into a fixed-size FPGA partition) through an abstraction called morphlets. Specifically, AMORPHOS virtualizes an FPGA as a set of morphable tasks, which can alter their resource requirements at runtime to dynamically accommodate a greater or lesser number of accelerators on the same FPGA. OPTIMUS does not support dynamic scalability on a single FPGA. However, since OPTIMUS supports acceleration preemption, OPTIMUS's virtual accelerators can theoretically be migrated in the event that a cloud provider wishes to alter an FPGA configuration.

The fundamental difference between AMORPHOS and OPTIMUS is that they target different FPGA platforms (host-centric vs shared-memory, respectively). The differences between these platforms are substantial (e.g., different software/hardware programming interfaces, memory latencies/-capacities, hardware topologies, and so forth).

Most importantly, these platforms necessitate significantly different forms of memory management. Because AMORPHOS targets host-centric platforms—where accelerators *cannot* issue their own DMAs—it focuses on virtualizing each accelerator's view of on-FPGA DRAM. Thus, AMORPHOS's memory protection logic only needs to manage on-FPGA DRAM, and can do so with segment-based translations.

On the other hand, OPTIMUS targets platforms in which the FPGA uses the system DRAM. Thus, OPTIMUS must integrate accelerator memory protection with the host's page-level memory management, while maintaining consistent views of each address space for the CPU and FPGA. Nonetheless, given that platforms such as Intel PAC [27] give FPGAs access to both system and on-FPGA DRAM, our approaches to memory virtualization are complementary to those of AMORPHOS.

7.2 Key Takeaways

We believe our work highlights two key areas for improvement in systems and architectural support for heterogeneous computing. First, there is a need for new OS abstractions. Currently, each FPGA vendor uses a different programming interface. Thus, standard OS abstractions (e.g., to send messages to the CPU and access different memories) would immensely increase program portability. FPGA manufacturers can hasten the arrival of such OS abstractions by providing a standardized hardware interface.

Second, a hard-wired multiplexer tree is needed to provide more efficient and scalable packet routing. Like AMORPHOS, OPTIMUS also confirms that a flat multiplexer becomes a bottleneck for scalability. Furthermore, OPTIMUS shows that even a programmer-synthesized multiplexer tree can be a bottleneck at higher frequencies. These bottlenecks arise due to the difficulty of placing multiplexer resources sufficiently close to pass timing constraints, but could be mitigated via a hard-wired multiplexer tree.

8 Related Work

Accelerator Libraries Amazon F1 [6] and Microsoft Brainwave [14, 50] offer accelerator libraries to their customers. The customer chooses from among these accelerators, ultimately running their acceleration job on an FPGA that has been configured accordingly. OPTIMUS is targeted for this use case, and allows the cloud provider to spatially and temporally multiplex their FPGAs among customers.

Sharing On-FPGA Memory Asiatici et al. propose a hypervisor featuring a high-level framework to facilitate FPGA application development [8]. The hypervisor provides a framework to share on-FPGA memory among multiple accelerators. CoRAM [19] and CoRAM++ [75] similarly allow software to read and write on-FPGA BRAMs. Unlike OPTIMUS, none of these designs grant the CPU and FPGA a unified view of memory.

Sharing System Memory FPGAs can share system memory with the CPU on platforms such as Intel PAC [27] (PCIe-only), Intel HARP [25] (PCIe and UPI), and Enzian [9] (forthcoming). GPUs from Intel [30, 70] and NVIDIA [1, 47, 58, 88] can transparently share memory regions with the CPU, using both software-only and hardware-assisted techniques. OPTIMUS's page table slicing is inspired by such GPU page table partitioning techniques (as well as those of Virtual WiFi [78]) in a hardware-software co-design that is independent of accelerator design and behavior.

Overlays FPGA overlays [13, 34, 35, 42] provide an abstraction of FPGA hardware such that configurations can be made architecture-agnostic. Unfortunately, the abstractions of overlays sacrifice throughput and resource utilization compared to configurations built for specific FPGA architectures. Given that the burden of developing accelerators is not placed on the customer in OPTIMUS, we believe that cloud providers and customers would prefer the efficiency of native builds over the ease of cross-platform porting.

Virtualizing FPGA Pools Xilinx SDAccel [82], Tarafdar et al. [67], and Microsoft Catapult [16, 56] target the virtualization of FPGA pools, allowing jobs to be scheduled on available accelerators within the pool. Unlike these systems, OPTIMUS targets the virtualization of individual FPGAs.

Virtualizing Individual FPGAs Prior work explores spatial multiplexing [15, 18, 53, 55, 72, 74] and temporal multiplexing [18, 53, 55, 73, 84] of FPGAs. While most of these works focus on host-centric FPGAs, OPTIMUS focuses on shared-memory FPGAs. An exception is AvA [84], which uses API remoting to virtualize accelerators. Unlike OPTIMUS, AvA targets a higher level of abstraction (e.g., OpenCL), and virtualizes the userspace library instead of low-level hardware.

FPGA OSes BORPH [64, 65] supplements software processes with *hardware processes*, which communicate with other processes via standard UNIX interfaces. ReconOS [48] and Hthreads [54] extend the domain of multi-threaded programming to an FPGA, and provide support for inter-thread communication and synchronization. LEAP [22] offers reliable and latency-insensitive communication channels between different hardware modules. AMORPHOS [37] provides support for sharing different on-FPGA resources. Unlike these works, OPTIMUS is a hypervisor that focuses on virtualizing shared-memory FPGAs as a set of accelerators.

SR-IOV for FPGAs Intel [26] and Xilinx [80] both offer IP to support hardware-assisted FPGA virtualization of PCIe transactions via SR-IOV [43]. However, state-of-the-art shared-memory FPGA platforms that use SR-IOV do not support more than one VF [25, 27]. OPTIMUS supports up to eight physical accelerators, which can each support both UPI and PCIe transactions as well as an arbitrary number of virtual accelerators.

Partial Reconfiguration A number of FPGA virtualization solutions [15, 18, 37, 74] target partial reconfiguration capabilities of FPGAs, where an individual accelerator can be reconfigured without needing to reconfigure the entire FPGA. Because Intel HARP currently only provides a single reconfigurable region on the FPGA, OPTIMUS does not support partial reconfiguration; doing so would overwrite the hardware monitor.

9 Conclusion

In this paper, we presented OPTIMUS, the first scalable hypervisor for shared-memory FPGA platforms. OPTIMUS provides both spatial and preemptive temporal multiplexing of FPGAs, such that individual accelerators on an FPGA can be fairly overprovisioned to guests. OPTIMUS offers efficient virtual DMA isolation via page table slicing. Our experiments show that OPTIMUS can support eight physical accelerators on a single FPGA, and improves the aggregate throughput of twelve realistic benchmark workloads by 1.98x-7x.

Acknowledgments

We thank our shepherd, Christopher J. Rossbach, and anonymous reviewers for their insightful feedback. This work was supported by Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. The testbed was supported in part by Intel Corporation and National Key Research & Development Program of China (No. 2016YFB1000502).

References

[1] [n.d.]. GP100 Pascal Whitepaper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[2] [n.d.]. Hugetlbfs Reservation. https://www.kernel.org/doc/html/v4.18/vm/hugetlbfs_reserv.html.

[3] [n.d.]. Open-Source FPGA Bitcoin Miner. https://github.com/progranism/Open-Source-FPGA-Bitcoin-Miner.

[4] [n.d.]. Transparent huge pages in 2.6.38. https://lwn.net/Articles/423584/.

[5] [n.d.]. Transparent Hugepage Support. https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

[6] Amazon. [n.d.]. Amazon EC2 F1 Instances - Run Customizable FPGAs in the AWS Cloud. https://aws.amazon.com/ec2/instance-types/f1.

[7] Amazon. [n.d.]. Official repository of the AWS EC2 FPGA Hardware and Software Development Kit. https://github.com/aws/aws-fpga.

[8] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A Fahmy, and Paolo Ienne. 2017. Virtualized execution runtime for fpga accelerators in the cloud. *Ieee Access* 5 (2017), 1900–1910.

[9] Systems Group at ETH Zurich. [n.d.]. Enzian is a research computer built by the Systems Group at ETH Zurich. http://www.enzian.systems/.

[10] Osama G Attia, Tyler Johnson, Kevin Townsend, Philip Jones, and Joseph Zambreno. 2014. Cygraph: A reconfigurable architecture for parallel breadth-first search. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, 228–235.

[11] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2018. Mosaic: Enabling Application-Transparent Support for Multiple Page Sizes in Throughput Processors. *SIGOPS Oper. Syst. Rev.* 52, 1 (Aug. 2018), 27–44. https://doi.org/10.1145/3273982.3273986

[12] Jayaram Bhasker. 1999. *A Vhdl Primer*. Prentice-Hall.

[13] Alexander Brant and Guy GF Lemieux. 2012. ZUMA: An open FPGA overlay architecture. In *2012 IEEE 20th international symposium on field-programmable custom computing machines*. IEEE, 93–96.

[14] Doug Burger. 2017. Microsoft unveils Project Brainwave for real-time AI. *Microsoft Research, Microsoft* 22 (2017).

[15] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. 2014. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. IEEE, 109–116.

[16] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. 2016. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 7.

[17] Ciro Ceissler, Ramon Nepomuceno, Marcio Pereira, and Guido Araujo. 2018. Automatic Offloading of Cluster Accelerators. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 224–224.

[18] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*. ACM, 3.

[19] Eric S Chung, James C Hoe, and Ken Mai. 2011. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 97–106.

[20] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (Jan 1998), 46–55. https://doi.org/10.1109/99.660313

[21] Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. 2015. Virtualized FPGA accelerators for efficient cloud computing. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*. IEEE, 430–435.

[22] K. Fleming, H. Yang, M. Adler, and J. Emer. 2014. The LEAP FPGA operating system. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.1109/FPL.2014.6927488

[23] Gokul Govindu, Ronald Scrofano, and Viktor K Prasanna. 2005. A library of parameterizable floating-point cores for FPGAs and their application to scientific computing. In *Proc Int'l Conf. Eng. Reconfigurable Systems and Algorithms (ERSA'05)*. Citeseer.

[24] Intel. [n.d.]. Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl-ias-ccip.pdf.

[25] Intel. [n.d.]. Hardware Accelerator Research Program. https://software.intel.com/en-us/hardware-accelerator-research-program.

[26] Intel. [n.d.]. Intel Arria 10 Avalon-ST Interface with SR-IOV PCIe Solutions User Guide. https://www.altera.com/en_US/pdfs/literature/ug/ug_a10_pcie_sriov.pdf.

[27] Intel. [n.d.]. Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/acceleration-card-arria-10-gx.html.

[28] Intel. [n.d.]. Intel Virtualization Technology for Directed I/O. https://software.intel.com/sites/default/files/managed/c5/15/vt-directed-io-spec.pdf.

[29] Intel. [n.d.]. Open Programmable Acceleration Engine. https://opae.github.io/latest/index.html.

[30] Intel. 2017. Intel Open Source HD Graphics and Intel Iris Plus Graphics Programmer's Reference Manual for the 2016 - 2017 Intel Core Processors, Celeron Processors, and Pentium Processors based on the "Kaby Lake" Platform. https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-kbl-vol05-memory_views.pdf.

[31] Intel. 2019. Embedded Peripherals IP User Guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_embedded_ip.pdf.

[32] Intel. 2019. Intel Arria 10 FPGAs. https://www.intel.com/content/www/us/en/products/programmable/fpga/arria-10.html.

[33] Intel. 2019. Intel FPGA Basic Building Blocks (BBB). https://github.com/OPAE/intel-fpga-bbb.

[34] Abhishek Kumar Jain, Suhaib A Fahmy, and Douglas L Maskell. 2015. Efficient Overlay architecture based on DSP blocks. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 25–28.

[35] Abhishek Kumar Jain, Douglas L Maskell, and Suhaib A Fahmy. 2016. Throughput oriented FPGA overlays using DSP blocks. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1628–1633.

[36] Neo Jia and Kirti Wankhede. [n.d.]. VFIO Mediated devices. https://www.kernel.org/doc/Documentation/vfio-mediated-device.txt.

[37] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 107–127.

[38] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Bedford Taylor. 2017. Moonwalk: NRE Optimization in ASIC Clouds. *SIGPLAN Not.* 52, 4 (April 2017), 511–526. https://doi.org/10.1145/3093336.3037749

[39] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux Virtual Machine Monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS)*.

[40] Oliver Knodel, Paul R Genssler, and Rainer G Spallek. 2017. Virtualizing Reconfigurable Hardware to Provide Scalability in Cloud Architectures. *Reconfigurable Architectures, Tools and Applications,*

RECATA (2017).

[41] Oliver Knodel and Rainer G Spallek. 2015. RC3E: provision and management of reconfigurable hardware accelerators in a cloud environment. *arXiv preprint arXiv:1508.06843* (2015).

[42] Dirk Koch, Christian Beckhoff, and Guy GF Lemieux. 2013. An efficient FPGA overlay for portable custom instruction set extensions. In *2013 23rd international conference on field programmable logic and applications*. IEEE, 1–8.

[43] Patrick Kutch. 2011. Pci-sig sr-iov primer: An introduction to sr-iov technology. *Intel application note* (2011), 321211–002.

[44] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. 2016. Coordinated and efficient huge page management with ingens. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 705–721.

[45] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. 2017. Ingens: Huge Page Support for the OS and Hypervisor. *ACM SIGOPS Operating Systems Review* 51, 1 (2017), 83–93.

[46] Doug Lea. [n.d.]. A Memory Allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[47] W. Li, G. Jin, X. Cui, and S. See. 2015. An Evaluation of Unified Memory Technology on NVIDIA GPUs. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 1092–1098. https://doi.org/10.1109/CCGrid.2015.105

[48] Enno Lübbers and Marco Platzner. 2009. ReconOS: Multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS)* 9, 1 (2009), 8.

[49] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. 2019. MEGA: overcoming traditional problems with OS huge page management. In *Proceedings of the 12th ACM International Conference on Systems and Storage*. ACM, 121–131.

[50] Microsoft. 2019. What are FPGAs and Project Brainwave? https://docs.microsoft.com/en-us/azure/machine-learning/service/concept-accelerate-with-fpgas.

[51] David Mulnix. 2017. Intel Xeon processor scalable family technical overview.

[52] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. 2017. Centaur: A framework for hybrid CPU-FPGA databases. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE, 211–218.

[53] Michele Paolino, Sébastien Pinneterre, and Daniel Raho. 2017. FPGA virtualization with accelerators overcommitment for Network Function Virtualization. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 1–6.

[54] Wesley Peck, Erik Anderson, Jason Agron, Jim Stevens, Fabrice Baijot, and David Andrews. 2006. Hthreads: A computational model for reconfigurable devices. In *2006 International Conference on Field Programmable Logic and Applications*. IEEE, 1–4.

[55] Sébastien Pinneterre, Spyros Chiotakis, Michele Paolino, and Daniel Raho. 2018. vFPGAmanager: A virtualization framework for orchestrated FPGA accelerator sharing in 5G cloud environments. In *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*. IEEE, 1–5.

[56] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 13–24.

[57] W. Qiao, J. Du, Z. Fang, M. Lo, M. F. Chang, and J. Cong. 2018. High-Throughput Lossless Compression on Tightly Coupled CPU-FPGA Platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 37–44. https://doi.org/10.1109/FCCM.2018.00015

[58] Nikolay Sakharnykh. 2018. EVERYTHING YOU NEED TO KNOW ABOUT UNIFIED MEMORY. http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf.

[59] Eric Schkufza, Michael Wei, and Christopher J Rossbach. 2019. Just-In-Time Compilation for Verilog: A New Technique for Improving the FPGA Programming Experience. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 271–286.

[60] Hardik Sharma, Jongse Park, Emmanuel Amaro, Bradley Thwaites, Praneetha Kotha, Anmol Gupta, Joon Kyung Kim, Asit Mishra, and Hadi Esmaeilzadeh. 2016. Dnnweaver: From high-level deep network models to fpga acceleration. In *the Workshop on Cognitive Architectures*.

[61] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[62] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 403–415.

[63] Any Silicon. [n.d.]. FPGA vs ASIC, What to Choose? https://anysilicon.com/fpga-vs-asic-choose/.

[64] Hayden Kwok-Hay So and Robert Brodersen. 2008. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 2 (2008), 14.

[65] Hayden Kwok-Hay So and Robert W Brodersen. 2007. *Borph: An operating system for fpga-based reconfigurable computers*. University of California, Berkeley.

[66] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. 2015. CAPI: A Coherent Accelerator Processor Interface. *IBM Journal of Research and Development* 59, 1 (Jan 2015), 7:1–7:7. https://doi.org/10.1147/JRD.2014.2380198

[67] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2017. Enabling flexible network FPGA clusters in a heterogeneous cloud data center. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 237–246.

[68] Terasic and Altera. [n.d.]. DE5a-Net FPGA Development Kit User Manual. https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-1804382103-de5a-net-user-manual.pdf.

[69] Donald Thomas and Philip Moorby. 2008. *The Verilog® Hardware Description Language*. Springer Science & Business Media.

[70] Kun Tian, Yaozu Dong, and David Cowperthwaite. 2014. A Full GPU Virtualization Solution with Mediated Pass-Through.. In *USENIX Annual Technical Conference*. 121–132.

[71] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. 2018. A survey on fpga virtualization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 131–1317.

[72] Duy Viet Vu, Oliver Sander, Timo Sandmann, Steffen Baehr, Jan Heidelberger, and Juergen Becker. 2014. Enabling partial reconfiguration for coprocessors in mixed criticality multicore systems using PCI Express Single-Root I/O Virtualization. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*. IEEE, 1–6.

[73] Wei Wang, Miodrag Bolic, and Jonathan Parri. 2013. pvFPGA: accessing an FPGA-based hardware accelerator in a paravirtualized environment. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*. IEEE, 1–9.

[74] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. 2015. 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, 1078–1086.

[75] Gabriel Weisz and James C Hoe. 2015. CoRAM++: Supporting data-structure-specific memory interfaces for FPGA computing. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–8.

[76] Gabriel Weisz, Joseph Melber, Yu Wang, Kermin Fleming, Eriko Nurvitadhi, and James C Hoe. 2016. A study of pointer-chasing performance on shared-memory processor-FPGA systems. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 264–273.

[77] Gabriel Weisz, Joseph Melber, Yu Wang, Kermin Fleming, Eriko Nurvitadhi, and James C. Hoe. 2016. A Study of Pointer-Chasing Performance on Shared-Memory Processor-FPGA Systems. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. ACM, New York, NY, USA, 264–273. https://doi.org/10.1145/2847263.2847269

[78] Lei Xia, Sanjay Kumar, Xue Yang, Praveen Gopalakrishnan, York Liu, Sebastian Schoenberg, and Xingang Guo. 2011. Virtual WiFi: bring virtualization from wired to wireless. In *Acm sigplan notices*, Vol. 46. ACM, 181–192.

[79] Xilinx. [n.d.]. AXI Interconnect. https://www.xilinx.com/products/intellectual-property/axi_interconnect.html.

[80] Xilinx. [n.d.]. Designing with SR-IOV Capability of Xilinx Virtex-7 PCI Express Gen3 Integrated Block. https://www.xilinx.com/support/documentation/application_notes/xapp1177-pcie-gen3-sriov.pdf.

[81] Xilinx. [n.d.]. DMA for PCI Express (PCIe) Subsystem. https://www.xilinx.com/products/intellectual-property/pcie-dma.html.

[82] Xilinx. [n.d.]. SDAccel Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[83] Peter Xu. 2018. Device Assignment with Nested Guest and DPDK. https://www.linux-kvm.org/images/a/a6/KVM_Forum_2018_viommu_vfio.pdf.

[84] Hangchen Yu, Arthur M. Peters, Amogh Akshintala, and Christopher J. Rossbach. 2019. Automatic Virtualization of Accelerators. In *17th Workshop on Hot Topics in Operating Systems (HotOS {XVII})*.

[85] H. Zeng, C. Zhang, and V. Prasanna. 2017. Fast Generation of High Throughput Customized Deep Learning Accelerators on FPGAs. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–8. https://doi.org/10.1109/RECONFIG.2017.8279792

[86] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. 2017. The Feniks FPGA Operating System for Cloud Computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*. ACM, 22.

[87] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. 2017. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 15–24.

[88] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 345–357.

[89] S. Zhou and V. K. Prasanna. 2017. Accelerating Graph Analytics on CPU-FPGA Heterogeneous Platform. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 137–144. https://doi.org/10.1109/SBAC-PAD.2017.25

A Artifact Appendix

A.1 Abstract

Here, we include links to our source code, and offer tutorials for installing the various components of OPTIMUS.

A.2 Artifact check-list (meta-information)

- **Program:** OPTIMUS hypervisor, hardware monitor, guest driver, guest core library, guest MPF library, and all benchmarks (including MemBench, LinkedList, Bitcoin, SSSP, and HardCloud applications).
- **Compilation:** GCC 4.8, Quartus Prime Pro 17.0.0.
- **Run-time environment:** CentOS 7.5 with Linux kernel version 5.1.0-rc6. When compiling the kernel, the configuration option FPGA_DFL must be disabled. QEMU 3.0.1.
- **Hardware:** Intel HARP platform (with Skylake CPUs, Arria 10 FPGAs, and Blue Bitstream version SR-6.4.0). Intel VT-d must be enabled in BIOS.
- **Publicly available?:** Yes

A.3 Description

A.3.1 How delivered:

All of the source code for OPTIMUS is open source, and can be obtained via GitHub[1] or Zenodo[2].

A.3.2 Hardware dependencies:

OPTIMUS requires an Intel HARP platform with Skylake CPUs, Arria 10 FPGAs, and Blue Bitstream SR-6.4.0. Intel VT-d (IOMMU support) must be enabled in the BIOS.

A.3.3 Software dependencies:

OPTIMUS requires GCC 4.8 to compile kernel modules and libraries and Quartus Prime Pro 17.0.0 for FPGA synthesis. Our experiment runs on CentOS 7.5 with Linux kernel 5.1.0-rc6 and QEMU 3.0.1. When compiling the kernel, the configuration option `FPGA_DFL` must be disabled.

A.4 Installation

Four components should be installed on the machine where OPTIMUS is deployed: (a) the OPTIMUS hypervisor, which is the key component of OPTIMUS and is used to provide FPGA virtualization support; (b) the host tools, which are used to configure and control the physical FPGA; (c) the guest driver, which is installed in guests and supports virtual accelerators; (d) the guest libraries, which help guest software use virtual accelerators.

The hardware monitor should be installed as a hardware library on the machine where FPGA bitstreams are synthesized. The hardware monitor is used to multiplex the FPGA interface and provide virtualization support.

The source code of these components can be found in our GitHub or DOI repository, we also provide a step-by-step tutorial (in our GitHub[1] or Zenodo[2] repositories) to help people who have a compatible HARP platform to build OPTIMUS from scratch.

A.5 Experiment workflow

There are four steps to run FPGA-accelerated applications in guest virtual machines: (a) synthesize a bitstream for a select group of accelerators, (b) install different components of OPTIMUS mentioned in A.4, (c) configure the FPGA with the bitstream synthesized in (a), and (d) boot guests and run different applications.

We provide the source code of our benchmarks as well as configuration files (which are used during synthesis) in our repository to help synthesize OPTIMUS compatible bitstreams.

A.6 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

[1] https://github.com/efeslab/optimus-hypervisor
[2] https://doi.org/10.5281/zenodo.3605682