A Fast and Flexible Hardware-based Virtualization Mechanism for Computational Storage Devices

Dongup Kwon, Dongryeong Kim, Junehyuk Boo, Wonsik Lee, and Jangwoo Kim,
Seoul National University

https://www.usenix.org/conference/atc21/presentation/kwon

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the 2021 USENIX Annual Technical Conference is sponsored by USENIX.

A Fast and Flexible Hardware-based Virtualization Mechanism for Computational Storage Devices

Dongup Kwon, Dongryeong Kim, Junehyuk Boo, Wonsik Lee, and Jangwoo Kim*
Department of Electrical and Computer Engineering
Seoul National University

Abstract

A *computational storage* device incorporating a computation unit inside or near its storage unit is a highly promising technology to maximize a storage server's performance. However, to apply such computational storage devices and take their full potential in virtualized environments, server architects must resolve a fundamental challenge: *cost-effective virtualization*. This critical challenge can be directly addressed by the following questions: (1) how to virtualize two different hardware units (i.e., computation and storage) and (2) how to integrate them to construct virtual computational storage devices, and (3) how to provide them to users. However, the existing methods for computational storage virtualization severely suffer from their low performance and high costs due to the lack of hardware-assisted virtualization support.

In this work, we propose *FCSV-Engine*, an FPGA card designed to maximize the performance and cost-effectiveness of computational storage virtualization. FCSV-Engine introduces three key ideas to achieve the design goals. First, it achieves high virtualization performance by applying *hardware-assisted virtualization* to both computation and storage units. Second, it further improves the performance by applying *hardware-assisted resource orchestration* for the virtualized units. Third, it achieves high cost-effectiveness by *dynamically constructing and scheduling* virtual computational storage devices. To the best of our knowledge, this is the first work to implement a hardware-assisted virtualization mechanism for modern computational storage devices.

1 Introduction

A modern *computational storage* device incorporating a computation unit inside or near its storage unit is becoming a highly promising solution for high-performance storage servers as it can minimize the data movement overhead with *near-storage processing* [10,15,17,21,27,33,35,39]. A server equipped with a computational storage device can offload data-intensive routines to the computation unit and enable it to access the data stored in the storage unit without software intervention. For example, current off-the-shelf computational storage devices incorporate high-end field-programmable gate arrays (FPGAs) for computations and adopt an NVM Express (NVMe) storage protocol to allow the FPGAs to directly access the solid-state drives (SSDs) [6,24,33,35].

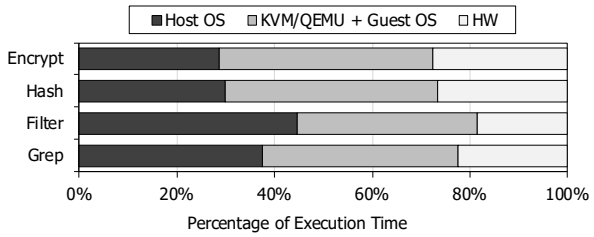


Figure 1: Percentage of execution time spent on a paravirtualized computational storage device.

In addition to the advances in the hardware devices, recent studies have proposed flexible FPGA overlay architectures and POSIX-like APIs to improve the usability of modern computational storage devices [33,35]. Their stream-based overlay architectures contain user-specified operators (e.g., stream processing units) and activate them selectively through an in-FPGA crossbar switch and control logic. At the same time, their custom software stacks provide abstraction layers to hide the complexity of the underlying FPGA implementations. For example, a recent computational storage implementation takes advantage of Linux file and pipe abstractions to allow users to easily orchestrate their near-storage processing [35].

However, the existing virtualization mechanisms for computational storage severely suffer from their low performance and high costs. First, the software-based virtualization mechanisms (e.g., paravirtualization) cannot take full advantage of near-storage processing due to their (1) heavy hypervisor and host OS stacks to emulate virtual computational storage devices and (2) indirect resource orchestration mechanisms via guest and host OSes. To profile the software overhead of paravirtualized computational storage devices, we measured the

*Corresponding author.

end-to-end execution time of near-storage processing benchmarks on a virtual machine (VM). The near-storage processing benchmarks read 4-KB pages from an NVMe SSD and perform four different stream operations (`encryption`, `hash`, `filter`, `grep`) on an FPGA. Figure 1 shows the percentage of execution time spent on the paravirtualized near-storage processing benchmarks. The hypervisor and guest/host OSes account for a significant portion of their overall execution time (72%–81%) due to the software-centric device emulation and resource orchestration.

Second, the existing virtualization mechanisms which statically allocate both computation and storage units for each VM will incur high hardware costs due to the inefficient use of the shared device resources. (1) The existing SSD-FPGA coupled architectures suffer from their limited scalability when multiple VMs share a single computational storage device. For example, concurrently running many VMs and their near-storage processing workloads on a single computational storage device result in the performance bottleneck at the single shared SSD. (2) Moreover, their static resource allocation methods cannot handle the dynamic behavior of VM workloads efficiently, which incurs the extra costs for the additional hardware resources to meet quality-of-service (QoS) requirements.

In this paper, we propose FlexCSV, a new hardware virtualization mechanism to maximize the performance and cost-effectiveness of computational storage virtualization. FlexCSV combines the following key ideas. First, FlexCSV implements *hardware-assisted virtualization and resource orchestration*. Through a standard single-root I/O virtualization (SR-IOV) layer at the hardware level, FlexCSV provides a fast and host-bypassing virtualization stack and allows multiple VMs to exploit near-storage processing capabilities. In addition, FlexCSV manages user-requested stream operations at the hardware level to mitigate the software burden to orchestrate near-storage processing.

Second, FlexCSV achieves high cost-effectiveness by *dynamically constructing and scheduling both computation and storage units*. To improve its scalability, FlexCSV adopts an SSD-FPGA decoupled architecture and allows the FPGA accelerator card to construct many virtual computational storage devices with multiple PCI Express (PCIe) attached SSDs. Moreover, its dynamic resource allocation from a shared hardware operator pool and partial reconfiguration support can capture the dynamic behavior of VM workloads and reduce QoS violations significantly at minimum hardware costs.

For evaluation, we implemented our FlexCSV prototype on a Xilinx FPGA accelerator card [7], NVMe SSDs [3], and an existing KVM/QEMU virtualization stack [4, 5]. We implemented a hardware-assisted virtualization stack for computational storage on the same FPGA card and connected its hardware modules through advanced extensible interface (AXI) interconnects. An in-FPGA AXI crossbar switch orchestrates data movements between the processing units and the FPGA's on-board DRAM. In this work, we implemented eight hardware near-storage processing operators and allowed them to be shared by four VMs.

Our experimental results show that our FlexCSV prototype obtains 2.0x–2.8x higher near-storage processing performance in virtualized environments than the existing software-centric virtualization mechanisms. Moreover, FlexCSV's SSD-FPGA decoupled architecture can connect four PCIe-attached SSDs and provide 3x more scalable performance over the coupled computational storage architectures. Through its dynamic resource allocation for both computation and storage units, FlexCSV can reduce QoS violations significantly at minimum hardware costs when a computation storage device is oversubscribed by many VMs.

In summary, we make the following contributions:

- **Novel virtualization stack for computational storage**: We propose a fast and flexible hardware-assisted virtualization mechanism for modern computational storage devices.
- **High performance**: FlexCSV achieves high virtualization performance by bypassing software stacks and leveraging near-storage processing.
- **High cost-effectiveness**: FlexCSV achieves high hardware cost-effectiveness by dynamically constructing and scheduling both computation and storage units at minimum costs.
- **Prototyping**: We implement and evaluate our FlexCSV prototype with off-the-shelf devices and open-source software virtualization stacks.

2 Background

2.1 SSD-FPGA Computational Storage

2.1.1 Hardware Architecture

SSD-FPGA hardware platform. A modern computational storage device incorporates its computation and storage units together and couples them through an on-board interconnect [24,33,35]. Figure 2 (bottom) shows the typical hardware platform of modern SSD-FPGA computational storage devices. It utilizes an FPGA for computations and allows it to directly access an attached NVMe SSD. For near-storage data processing, it also supports peer-to-peer (P2P) data communications between the computation and storage units through its internal switches (e.g., on-board PCIe switch). For example, by exposing an FPGA's DRAM space to an SSD and implementing a routing policy in an internal crossbar switch, a computational storage device can enable the computation and storage units to exchange data directly without software arbitration [35].

FPGA overlay architecture. The FPGA overlay architectures implemented on computational storage devices offer programmable near-storage processing [33, 35]. They consist

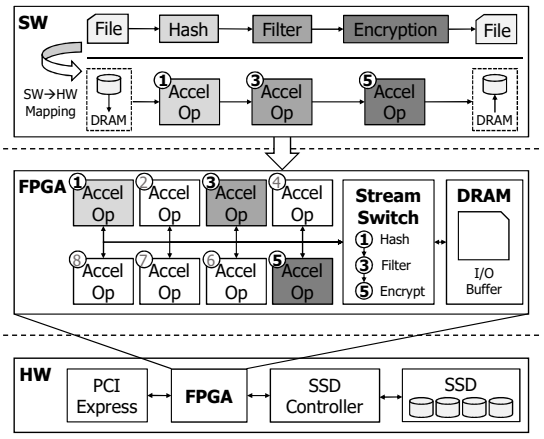


Figure 2: Full software and hardware stacks for a modern FPGA-based computational storage device.

```
typedef struct {
  ap_uint<64> data;
  ap_uint<4>  dest;
  ap_uint<1>  last;
} stream_data;
typedef hls::stream<stream_data> stream_t;

void accel_unit(
stream_t &in, stream_t &out, ap_uint<4> dest){
  stream_data input, output;
  do {
    input = in.read();
    output.data = process(input,data);
    output.dest = dest;
    out.write(output);
  } while (!input.last);
}
```

Listing 1: Example HLS code for an operator implemented on an FPGA overlay architecture.

of reconfigurable operators, crossbar stream switches, and on-board DRAM. The *operators* are reconfigurable stream processing units, and each of them can be selectively activated to configure full stream operations. The on-board crossbar switches orchestrate data movements between the operators and the FPGA's DRAM. Figure 2 (middle) shows an example overlay architecture for programmable near-storage processing. This overlay architecture contains eight operators and three of them are activated to serve the user-specified stream workloads (`hash`, `filter`, `encryption`). Then, a routing policy is installed in the stream switch to properly route an incoming data stream to the target operators (`hash`→`filter`→`encryption`) or DRAM buffers.

The FPGA overlay architectures also implement storage interfaces to allow their operators to directly access storage units [24,33,35]. Modern computational storage devices adopt an NVMe standard storage protocol to offer fast and parallel storage access [24]. NVMe has the following key advantages. First, NVMe supports multiple I/O queues to fully utilize the high-bandwidth storage units. NVMe can run multiple storage operations concurrently by assigning separate NVMe submission queue (SQ)/completion queue (CQ) pairs to different processing cores. Second, NVMe enables fast storage operations by minimizing the number of memory-mapped I/O (MMIO) operations. For example, NVMe devices expose a set of SQ/CQ doorbell registers and require only a single MMIO write operation (i.e., doorbell register write) to submit storage operations or to notify their completions.

2.1.2 Software Support

Abstraction layer. To improve the usability of near-storage processing in modern computational storage devices, recent studies provide custom software stacks utilizing Linux file and pipe abstractions and allow users to easily orchestrate near-storage processing [33,35]. Their software stacks expose the operators implemented on their FPGA overlay architectures as executable files to an OS. Then, a user program can initiate near-storage processing through POSIX-like APIs or a pipe command from data to executable (i.e., operator) files. Figure 2 (top) shows an example user program and its software-to-hardware mapping process. In this example, the application performs a `hash`→`filter`→`encryption` stream operation on the input file stored in the computational storage device. In this way, the software support can hide the complexity of the underlying FPGA implementations and coordinate user-specified near-storage data processing.

High-level synthesis support for operators. The software support also allows users to customize operators using an FPGA's reconfigurability and high-level synthesis (HLS) tools [35]. Listing 1 shows an HLS code snippet to implement an example `accel_unit` operator. First, users can define a stream data structure. In this example, `dest` and `last` signals are delivered along with a `data` stream. The `dest` field indicates its next destination operator and the `last` field indicates a last word in the `data` stream. Second, users can define the input and output ports of an operator. In this example, the `accel_unit` module has the `in`/`out` data stream ports and the `dest` configuration register to determine its next destination operator. This example operator reads a word through the `in` stream port and forwards it through the `out` stream port after manipulating the `dest` field of the output stream.

2.2 I/O Virtualization

2.2.1 Software-based Virtualization

The existing software-based I/O virtualization presents virtual device instances and enables device sharing across multiple VMs. *Full virtualization*, which is one of the software-based virtualization mechanisms, utilizes a trap-and-emulate approach to provide virtual device instances to VMs without changing the guest OSes. However, this virtualization mechanism significantly suffers from excessive VM exits when

guest OSes access their virtual device resources.

In contrast to full virtualization, *paravirtualization* enables efficient virtual device emulation by creating VM-friendly virtual device interfaces between guest OSes and hypervisors (e.g., virtio [34]). This paravirtualization mechanism incurs a fewer number of VM exits by minimizing MMIO operations for device access, but it still relies on software traps to a hypervisor and CPU mode switches. Moreover, guest OSes should be aware that they are being virtualized and modified to interact with a hypervisor in this efficient manner.

To virtualize modern fast and high-bandwidth devices, recent *sidecore approaches* dedicate multiple CPU cores for device emulation [29, 38]. As dedicated sidecores in the host software keep polling guest I/O operations via shared memory regions, VMs do not have to incur VM exits to submit device operations. In this way, the sidecore approaches minimize the performance overhead incurred by VM exits and CPU context switches [23]. However, the sidecore approaches demand a large amount of computing resources of a host server machine to execute their polling-based device emulation [22, 25, 29].

2.2.2 Hardware-assisted Virtualization

To overcome the performance overhead and host inefficiency of the software-based virtualization mechanisms, hardware-assisted virtualization mechanisms allow guest OSes to access target PCIe devices directly without any software arbitration. To enable the direct assignment of PCIe devices (i.e., *passthrough* virtualization), an I/O memory management unit (IOMMU) and its direct memory access (DMA) and interrupt remapping mechanisms are introduced. The DMA remapping mechanism allows DMA operations from virtual devices to be accomplished with guest physical addresses. Similarly, the interrupt remapping mechanism translates interrupt vectors caused by the virtualized devices into VM contexts. However, this approach requires a physical device to be exclusively assigned to a single VM and does not support device sharing across multiple VMs.

To address such shortcomings, PCIe SR-IOV allows a physical device to be shared by many VMs at the hardware level. An SR-IOV capable device presents multiple physical functions (PFs) and virtual functions (VFs) (i.e., virtual device instances) at the device interface. Since VFs have separate PCIe configuration registers, including base address registers (BARs), SR-IOV can enforce resource isolation while serving multiple VMs. Moreover, an SR-IOV capable device implements how to multiplex itself at its internal bridge module and thus does not rely on any host software to multiplex its virtual device instances.

3 Motivation

The increasing density and performance of modern computation and storage devices create new opportunities for virtualization. Figure 3 shows the increase in SSD capacity and FPGA resources over the recent eight years. The available commodity SSD and FPGA resources have increased 3x–6x over the years, which creates a high potential to serve many VMs on a single computational storage device [19, 22, 25, 29, 40]. Furthermore, the significant improvements in storage device access time (65 μs→5 μs [33]) have moved the performance bottlenecks to the software stacks.

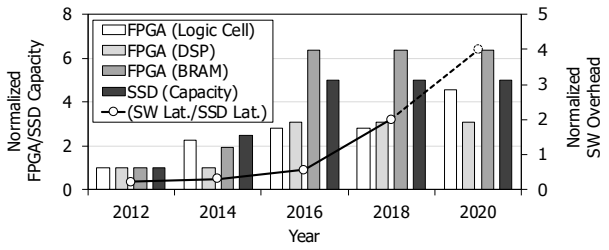
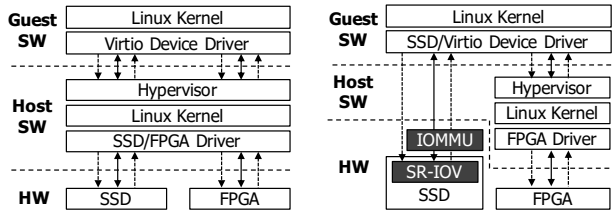


Figure 3: Increasing resource density and performance of FPGA and SSD products over time [3, 8]. The data are normalized to the values at 2012. The latency of the SW stack is set to 20 μs [30, 33], and the current SSD access time is assumed to be 5 μs based on recent studies [33, 41].

However, the existing software-based SSD and FPGA virtualization mechanisms have the following limitations when providing near-storage processing between the virtualized computation and storage units. First, the software-based virtualization mechanisms cannot take full advantage of near-storage processing due to their indirect device-control and data paths to emulate virtual computational storage devices. Second, their static resource allocation for each VM incurs high hardware costs due to the inefficient use of the shared device resources.

3.1 Indirect Device-Control and Data Paths

Employing software-centric virtualization to SSD and FPGA devices separately suffers from indirect device-control and data paths between the virtualized hardware units and fails to take full advantage of near-storage processing. To measure the software overhead of paravirtualized SSD and FPGA operations, we implemented a virtio-based virtualization stack on KVM/QEMU and profiled the end-to-end execution time of SSD-FPGA near-storage processing. Figure 4 shows two software-centric implementations for computational storage virtualization. In the full software implementation, SSD and FPGA operations involve VM exits and traps to a hypervisor. In the optimized software implementation, accessing an SSD from a VM can bypass the hypervisor and host OS stacks through to an IOMMU and SR-IOV support. However, utilizing an FPGA still relies on the software-centric virtualization mechanisms and suffers from the software-side performance overhead. Moreover, since a guest OS cannot obtain host physical addresses of the FPGA's BARs, its input and output data must be transferred via the guest and host OS stacks.



(a) Full-SW (paravirtualization) (b) SR-IOV (SSD) + SW (FPGA)

Figure 4: Software-centric virtualization mechanisms for computational storage.

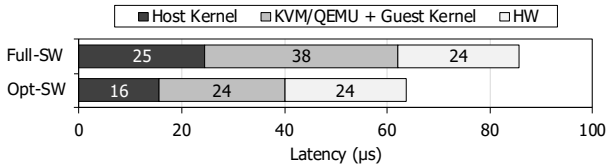


Figure 5: Execution time spent on a paravirtualized computational storage device with SW-centric virtualization implementations.

Figure 5 shows the execution time spent in paravirtualized SSD-FPGA near-storage processing. The benchmark reads a 4-KB page from the NVMe SSD and executes Blowfish encryption [28] on the FPGA. To emulate a coupled computational storage device, we installed an Intel Optane SSD [3] and Xilinx Alveo U250 FPGA [7] and connected them through PCIe Gen-3 lanes. The SSD read latency is 10 μs, and the encryption operation takes 14 μs to process the 4-KB data. In this environment, the hypervisor and guest/host OSes account for a significant portion of overall execution time due to the software-centric device emulation and resource orchestration. The optimized software virtualization implementation mitigates host OS overhead because the SSD access bypasses the host software. Also, as there is no need for KVM/QEMU to emulate the SSD, the hypervisor and guest kernel overhead also decrease. However, the overhead for the virtual FPGA emulation and the data movements between the virtualized units still remains.

3.2 SSD-FPGA Coupled Architecture

An SSD-FPGA coupled computational storage architecture severely suffers from its limited scalability due to its board-level SSD-FPGA integration [6, 17, 33, 35]. Such tight device integration makes it challenging to merge diverse SSD-FPGA resource combinations into a single device while supporting direct device-control and data paths among all the consolidated devices. Moreover, their architectural limitations become increasingly apparent as the gaps between SSD and FPGA resource capacity and performance increase.

For example, concurrently executing many VMs and their near-storage processing workloads on a computational storage device can lead to the performance bottleneck at the single shared SSD. To show the performance impact of resource contention in a coupled architecture, we emulated an on-board consolidated storage unit by dedicating an NVMe SSD to an FPGA board through PCIe P2P. In this experiment, we implemented an example operator whose throughput is 100 MB/s and measured the end-to-end latency on a target VM with the increasing number of I/O-intensive VMs and their block I/O intensities (i.e., queue depth). Figure 6 shows the normalized end-to-end latency of the target near-storage processing on the coupled architecture. When two or more I/O-intensive VMs share the SSD, the near-storage processing latency becomes 2.1x slower than the single-VM execution case. When the VMs demand higher I/O performance by increasing the queue depth, the target VM and its near-storage processing suffer from the more severe resource contention.

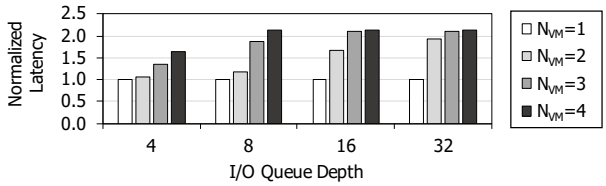


Figure 6: Performance comparison of near-storage processing with the increasing number of SSD-contending VMs and queue depth. The results are normalized to the single-VM latency. N_{VM} indicates the number of concurrently running VMs.

3.3 Static SSD/FPGA Resource Allocation

Static resource allocation and scheduling for both computation and storage units will incur high hardware costs because they cannot handle the dynamic behavior of VM workloads efficiently. To motivate dynamic resource allocation and scheduling, we implemented two hardware operators on an FPGA and allowed each of them to serve two VMs using time multiplexing. Following the resource-sharing mechanisms proposed by prior FPGA virtualization studies [19, 26], we oversubscribed the hardware operators but statically assigned them to a specific set of VMs.

Figure 7 shows the latency cumulative distribution function of the four VM workloads with the static resource allocation strategy. In this experiment, we executed four VMs concurrently and generated VM workloads by following Poisson distribution with different expected request rates (λ). For the first two VMs (VM_1, VM_2), we generated near-storage processing workloads with the same execution time and wait time ($T_{exec} = T_{wait}$, $\lambda_A = \frac{1}{T_{exec}+T_{wait}} = \frac{1}{2T}$). For the other two VMs (VM_3, VM_4), we generated workloads with a longer period between near-storage processing invocations ($\lambda_B = \frac{1}{16T}$). The result demonstrates that the static operator allocation scheme cannot guarantee a target QoS with the skewed workloads. When the VM_1 and VM_2 share the same operator and invoke

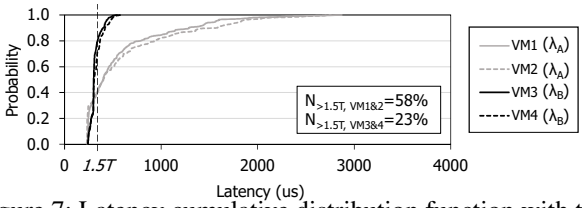


Figure 7: Latency cumulative distribution function with two oversubscribed hardware operators and four VMs.

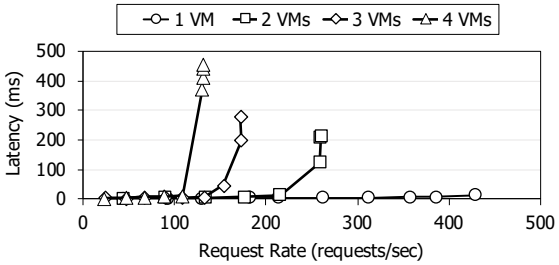


Figure 8: Operator latency with the increasing number of VMs and their request rate.

near-storage processing with a high request rate (λ_A), they severely suffer from a larger number of QoS violations ($> 1.5 \times T_{exec}$) than the other VM group (λ_B) due to the severe resource contention between the VMs.

The static resource allocation and scheduling mechanisms can lead to unacceptable QoS levels with the increasing number of VMs. Figure 8 shows the average latency of an encryption operator with varying request rates. If the total request rate does not exceed the maximum throughput of the operator, the operator can serve the requests within a reasonable latency bound. For example, when the four VMs submit requests at a rate of 50 requests per second each, the operator will show reasonable latency bound and the VMs will not suffer from the unexpected delay. However, when the total request rate exceeds the maximum throughput, the operator latency and the number of QoS violations increase quickly.

4 Design and Implementation

4.1 FlexCSV Architecture

In this work, we propose FlexCSV, a fast and cost-effective virtualization mechanism for computational storage devices. Its key idea is to implement *FCSV-Engine*, an FPGA card designed to maximize the performance and cost-effectiveness of computational storage virtualization. Figure 9 shows the FCSV-Engine architecture and its main hardware components. FCSV-Engine implements (1) a hardware-assisted virtualization layer based on PCIe SR-IOV, (2) a hardware-level direct device-orchestration mechanism, (3) an SSD-FPGA decoupled architecture, and (4) dynamic resource allocation through its operator renaming and partial reconfiguration support.

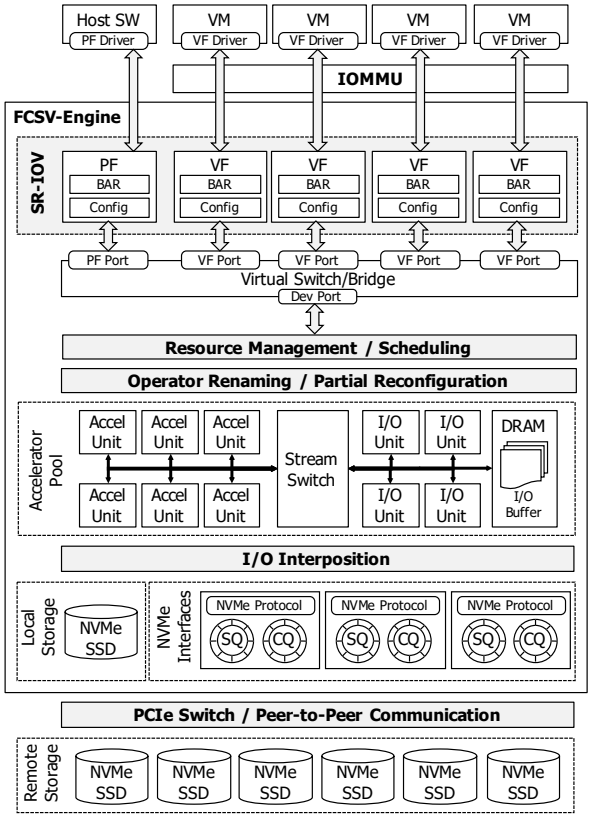


Figure 9: FCSV-Engine architecture.

4.2 Hardware-assisted Virtualization

4.2.1 SR-IOV and VM Isolation

To mitigate the software overhead in computational storage virtualization, FCSV-Engine offers hardware-assisted virtualization under a standard PCIe SR-IOV layer. By incorporating SR-IOV, FCSV-Engine can virtualize itself at the hardware level and each VF can be assigned exclusively to a VM for the direct access. Another advantage is that SR-IOV does not demand extra server CPU cores for polling guest I/O activities and indirect interrupt injections, which enables even more scalable and cost-effective server configurations than the conventional software-centric virtualization mechanisms (e.g., trap-and-emulate, sidecore). In this work, we utilize a single SR-IOV implementation at FCSV-Engine to virtualize both computation and storage units. This design choice minimizes the server costs for purchasing and operating SR-IOV supported computation and storage devices.

In addition, we utilize non-overlapping address translation to FCSV-Engine's internal address space (i.e., PCIe-to-AXI address translation) to guarantee isolated execution of multiple VMs. FCSV-Engine allocates a disjoint set of memory regions and assigns different AXI address ranges for each VF so that near-storage processing requests from two different VMs do not interfere with each other. For example, we statically partition FCSV-Engine's on-chip memory and off-chip

DRAM regions, and apply different offset values to MMIO requests from the VMs. Similarly, we allocate a different set of available interrupt vectors of FCSV-Engine for each VM. In this work, we allocate eight interrupt vectors per VM, and each vector is dedicated to a single completion queue of FlexCSV device driver running on a guest OS.

4.2.2 Device Interface

FCSV-Engine implements a multi-queue device interface and a doorbell mechanism to interact with guest OSes. For this multi-queue device interface, we arrange FCSV-Engine's PCIe BAR regions for doorbell registers of virtual FCSV-Engine instances (i.e., each VF). From the software side, FCSV-Engine's device driver installed on a guest OS allocates multiple SQ/CQ pairs and initializes the doorbell registers mapped at FCSV-Engine's BARs. FCSV-Engine's device driver then delivers the guest physical addresses of the allocated queue pairs to FCSV-Engine. The number of queue pairs is dictated by FCSV-Engine's BAR configuration and FPGA on-chip resource budgets. In this work, we create eight queue pairs per VM so that the same number of virtual CPUs can offload near-storage processing in parallel.

FCSV-Engine polls its on-chip memory space for doorbell registers using its host interfaces. To serve near-storage processing requests from multiple VMs concurrently, we instantiate multiple host interfaces and dedicate them to each VF. They get newly updated doorbell values by polling the doorbell register regions and utilize an internal DMA engine and an IOMMU to access a target SQ in guests' memory space. By incorporating an IOMMU and its guest-to-host address translation mechanism, each virtual FCSV-Engine instance can safely access target queue pairs allocated in the guest memory space without software intervention. Alternatively, FCSV-Engine can manage a guest-to-host address mapping table in itself, but this design incurs significant memory overhead to store the translation tables for every VM.

4.3 Hardware-level Resource Orchestration

4.3.1 Near-storage Processing Command

To offload resource orchestration routines to FCSV-Engine, FlexCSV extends a standard NVMe protocol and defines a new command format for near-storage processing. FlexCSV's NVMe-extended computational storage protocol minimizes software modifications to support near-storage processing in virtualized environments. Since major cloud providers are allowing NVMe storage devices to be used as primary storage for VMs, our NVMe-extended protocol can be easily applied in modern cloud and datacenter infrastructures.

Figure 10 shows a near-storage processing command structure. First, `op_chain` specifies which operators should be activated and the target stream order of the activated operators. For this, every operator type implemented on an FPGA

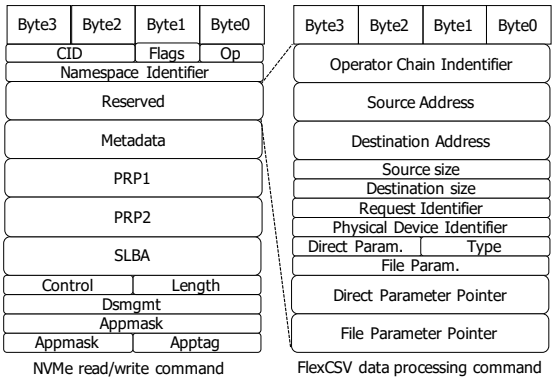


Figure 10: Near-storage processing command in FlexCSV.

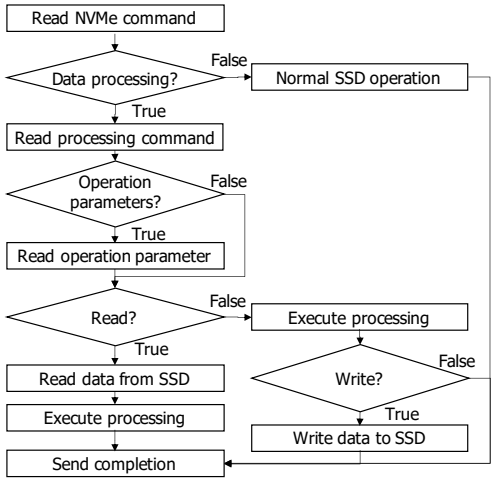
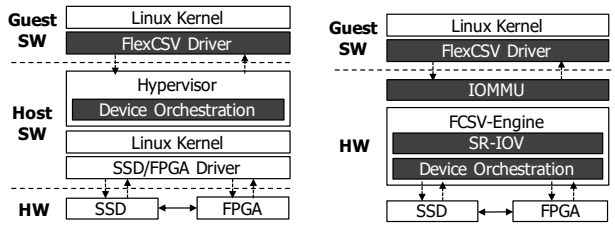


Figure 11: Control flow in FCSV-Engine.

has a unique identifier. `src_addr/size` and `dst_addr/size` represent the addresses and sizes of the source and destination in the FPGA's DRAM space. Additionally, if an operator needs parameters to process, a user can carry the parameters either directly or indirectly with `direct_param` or `file_param` fields. A user can also manage dependency between near-storage processing commands by manipulating a `request_id` field. If a request contains the same `rid` as in the previous requests, the current request cannot be issued before the earlier requests finish their near-storage processing. `type` determines whether a current request involves storage access or not.

4.3.2 Resource Orchestration

FCSV-Engine involves a resource orchestration mechanism to manage both computation and storage resources. To orchestrate two different hardware units without frequent software-hardware crossings, FCSV-Engine schedules user-requested computation and storage operations at the hardware level. Figure 11 shows the hardware-level scheduling mechanism. First, FCSV-Engine identifies a newly issued command by polling submission doorbell registers. If the command re-



(a) SW FCSV-Engine (b) HW FCSV-Engine

Figure 12: FCSV-Engine implemented in SW and HW.

quires near-storage processing, FCSV-Engine reads the data processing command and saves its contexts (e.g., source/destination FPGA DRAM addresses). After that, FCSV-Engine manipulates the DMA buffer addresses of the received NVMe command so that the SSD can transfer the user-requested data to/from the FPGA's DRAM. FCSV-Engine also enforces a correct order of computation and storage operations depending on the direction of user-requested data movements (i.e., read or write) and target operators.

FCSV-Engine executes user-requested stream data processing without software orchestration. In contrast to the existing static routing mechanisms, FCSV-Engine adopts a dynamic routing mechanism between hardware operators. For this, FCSV-Engine's control logic determines the order and routing path dynamically when it executes near-storage processing. FCSV-Engine parses the `op_chain` field of a data processing command and reserves the shortest routing path by manipulating the `dest` configuration registers of the user-requested operators. The operators then manipulate the `dest` field of the output stream data and FCSV-Engine's internal crossbar switch redirects the incoming stream to the correct next operator. If the requested operators are used and their paths are already reserved, FCSV-Engine stalls their executions until the earlier requests finish their near-storage processing.

4.3.3 Software FCSV-Engine

FCSV-Engine's resource orchestration mechanism can be implemented at the hypervisor level, but it still suffers from frequent software-hardware layer crossings to orchestrate two different hardware units. Figure 12 illustrates the software and hardware FCSV-Engine implementations. In the software implementation, a guest OS can leverage an NVMe-extended near-storage processing protocol and thus reduce the number of guest-host layer transitions. It also allows a device to directly transfer data to another device's internal memory by manipulating the DMA buffer addresses to the target device memory addresses. However, this design suffers from the inevitable hypervisor and host OS overhead due to the indirect orchestration for SSD and FPGA devices and a large number of software-hardware layer transitions.

Because of the indirect resource orchestration routines through the host software (e.g., MMIO, interrupt), the software FCSV-Engine implementation fails to achieve the full potential of near-storage processing [21]. To profile the performance overhead of the software FCSV-Engine implementation, we measured the SSD access latency through (1) the host software, (2) PCIe P2P, and (3) on-chip interconnect (ICN). Figure 13 shows their round-trip time for access to an Intel Optane SSD. The result indicates that the major performance bottleneck is moved to the software stacks when the host software manages a device operation. Such performance overhead will get worse when we exploit near-storage processing between faster computation and storage units. On the other hand, the round-trip time of the PCIe P2P and on-chip ICN is still much faster than the software MMIO and interrupt mechanisms.

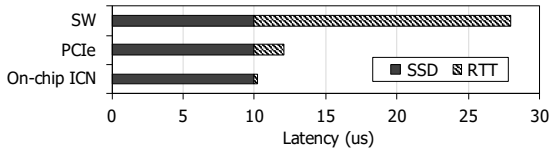


Figure 13: Storage latency and round-trip time through SW, PCIe, and on-chip ICN.

4.4 SSD-FPGA Decoupled Architecture

In this work, we introduce a decoupled computational storage architecture for scalable near-storage processing with multiple PCIe-attached SSDs. To allow multiple NVMe SSDs to combine with FCSV-Engine through PCIe P2P, the host software remaps their queue pairs onto FCSV-Engine's BAR regions. By doing so, the decoupled storage units can seamlessly exchange NVMe commands and their completions with FCSV-Engine. The SSDs are unaware of being interacting with FCSV-Engine, but an external PCIe switch delivers their PCIe read and write transactions to FCSV-Engine directly. In addition, FlexCSV can nicely scale with a large number of PCIe-attached SSDs leveraging its large on-chip memory space. As a result, FlexCSV can mitigate the performance bottleneck at a single FPGA or SSD by flexibly combining PCIe-attached computation and storage devices in the same server.

Moreover, FCSV-Engine implements PCIe message arbitration and transaction modules to encapsulate local NVMe requests (e.g., NVMe doorbell write) with PCIe transactions and allow multiple storage interfaces to share a single PCIe/DMA IP core. In this arbitration module, FCSV-Engine leverages PCIe transaction queues for each VM and adopts a round-robin algorithm to serve PCIe access from multiple VMs. The PCIe transaction module translates an internal AXI address to an associated host physical address (e.g., AXI-to-PCIe address translation). For this address translation, the PCIe/DMA IP core manages an AXI-to-PCIe address mapping table. The current Xilinx-provided PCIe/DMA IP core supports up to six mapping entries, and this design point can be a serial

	Critical Path		FPGA Resource Usage		
	Submission	Completion	LUT	FF	BRAM
Static	656 ns	72 ns	20370	6702	11
Renaming	728 ns	96 ns	20693	6892	11

Table 1: Renaming overhead analysis.

point when it handles concurrent near-storage processing from many VMs.

4.5 Dynamic Resource Allocation

4.5.1 Operator Renaming

To maximize the hardware resource efficiency, FCSV-Engine implements dynamic resource allocation through its operator renaming support. FCSV-Engine implements a shared operator pool and dynamically maps user-requested operations onto available physical operators. In this way, FCSV-Engine can quickly capture the dynamic behavior of VM workloads and thus reduce QoS violations significantly. Our current resource scheduling mechanism is similar to a resource availability-based FCFS algorithm because we focus more on cost-effectiveness and resource utilization of computation and storage units. However, we can also improve storage fairness and performance by adopting more fairness-oriented scheduling methods [16, 36].

In the renaming stage, every operator request from VMs is mapped onto physical operators via an operator renaming table. The operator renaming table manages the availability of physically implemented operators. FCSV-Engine looks up the renaming table to find and allocate available physical instances of user-requested operators. If it succeeds in allocating the physical operators, the scheduler executes near-storage processing and manipulates the operator renaming table to record the resource allocation status. When the requested near-storage processing finishes, the scheduler collects the completions from all the activated operators and deallocates the recorded resources by manipulating the renaming table.

We measured the area and performance overhead when the operator renaming mechanism is implemented in FCSV-Engine's scheduler module. We first implemented the scheduler without the renaming capabilities in which virtual operators have a one-to-one mapping with physical operators, and implemented operator renaming logic on top of it. In this work, our operator renaming logic can remap a virtual operator onto four physical operator instances. Table 1 shows the increase in the area and critical path for our operator renaming logic. The area overhead is 2%–4% and the total critical path overhead is around 100 ns, which is negligible compared to the original scheduler area and operator delay.

4.5.2 Operator Partial Reconfiguration

Partial reconfiguration (PR) support for FPGA operators further improves the hardware utilization by capturing the dynamic behavior of VM workloads. If there exist enough operator slots to serve all requested near-storage processing workloads, their target operators can be assigned to available slots in FCSV-Engine's operator pool. However, if the demand from near-storage processing workloads exceeds the maximum number of operator slots, they can be partially reconfigured to serve the current user requests. As a result, such a dynamic resource allocation mechanism enables the FPGA to support more operators than its physical resource limit.

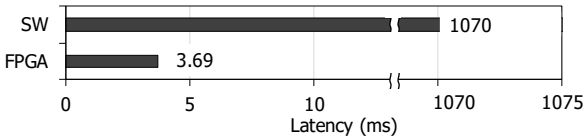


Figure 14: Partial reconfiguration latency with the host software and an FPGA.

	CLB LUTs	CLB Registers	Block RAM Tiles	Clock speed
FCSV-Engine	313,069 (18%)	487,333 (14%)	441 (16%)	250 MHz
PCIe / SR-IOV	46,970	51,487	61	
DRAM	18,592	20,817	26	
Host Interface (4 VMs)	74,886	124,761	266	
Storage Interface	22,927	39,403	39	
Scheduler	2,507	3,050	4	
Interconnect	115,073	203,827	0	
Others	32,114	43,988	45	

Table 2: FCSV-Engine FPGA resource utilization.

Figure 14 shows the latency of partially reconfiguring the FPGA operators from the host software and from the FPGA itself. Reconfiguration from the host software incurs long latency because the software has to send the bitstream from the host DRAM to the FPGA. Alternatively, to achieve agile operator reconfiguration, we can store the partial bitstreams in the FPGA DRAM and implement a PR controller to allow the FPGA to reconfigure its operator slots. In this way, the PR latency is reduced by 99.7% and the scheduler in FCSV-Engine can dynamically generate the operators on demand with low overhead.

5 Evaluation

5.1 Experimental Setup

To evaluate FlexCSV, we implemented our FCSV-Engine prototype on a Xilinx Alveo U250 board and installed its SW support on the Linux KVM/QEMU virtualization stack. Our FCSV-Engine prototype provides eight stream-based operators with the partial reconfiguration support and allows the operators to be shared by four VMs through the hardware-assisted virtualization and resource allocation mechanisms.

Operators	LUTs	Registers	BRAMs	Performance
Encryption (E) [28]	38,031	19,246	168	211 MB/s
Decryption (D) [28]	37,655	19,126	168	212 MB/s
Hash (H) [37]	50,620	13,541	0	285 MB/s
Aggregate (A) [35]	2,292	1,539	2	4.70 GB/s
Filter (F) [35]	41,823	5,428	116	278 MB/s
Grep (G) [15]	37,288	5,647	6	426 MB/s
KNN (K) [32]	14,052	4,091	8	4.22 GB/s
Bitmap (B) [33]	63,570	5,588	31	4.24 GB/s

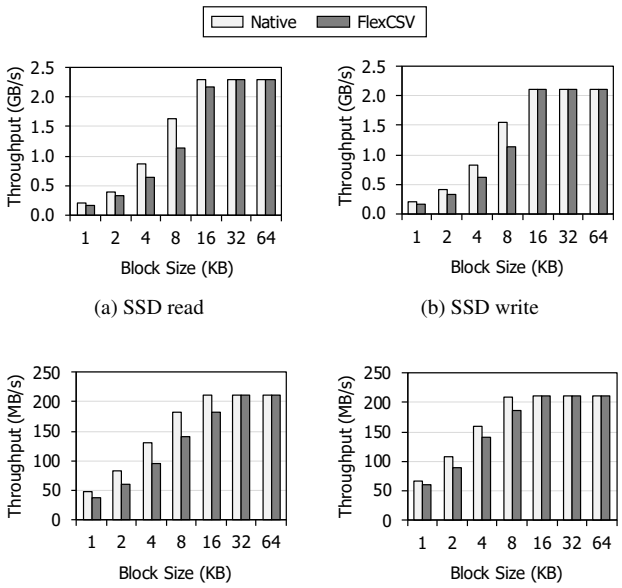
Table 3: Operators' FPGA resource utilization.

To enable FCSV-Engine to interact with CPUs and PCIe-attached NVMe SSDs, we utilized Xilinx-provided PCIe and DMA engine implementations (PCIe Gen3 4-lane, 4 GB/s per direction) [9] and configured VFs to virtualize FCSV-Engine itself. For intermediate data buffers between computation and storage units, FCSV-Engine leverages its on-board DDR4 DRAM and on-chip AXI-stream FIFO queues.

Table 2 shows FCSV-Engine's FPGA resource utilization using Xilinx Vivado and HLS (v2019.2). The on-chip interconnect logic consumes the major portion of FPGA LUTs and registers as it connects all the VFs (i.e., host interfaces for each VM) and operators through all-to-all crossbars. On the other hand, the host and storage interfaces consume many on-chip memory tiles for their device register regions of the NVMe-extended protocol (e.g., doorbell registers) and for NVMe I/O queue regions to orchestrate PCIe-attached NVMe SSDs. Note that the FPGA has enough remaining resources to add more operator slots for near-storage processing.

Our software-hardware full-system prototype is built on a host server with two Intel's Xeon Gold 5118 CPUs, each with 12 physical cores running at 2.3 GHz, and 256-GB DDR4 DRAM (Supermicro SuperServer 4029GP-TRT2). The host server machine is equipped with four Intel Optane 900P NVMe SSDs and connects them to FCSV-Engine through PCIe Gen3 4-lane interconnects (4 GB/s per direction). The Optane SSD can offer up to 550k IOPS in random-read and 500k IOPS in random-write with 10 μs latency. For software support, we installed a Ubuntu 18.04 OS and Linux kernel (version 5.3) on VMs and implemented custom FCSV-Engine's device driver.

To generate various near-storage processing scenarios, we implemented eight representative computational storage operations from the previous FPGA acceleration and near-storage processing studies [15, 28, 32, 33, 35, 37]. We utilized the Xilinx HLS tool (v2019.2) to generate their hardware operators. The implemented hardware operators perform stream operations through 512-bit input and output data links (16 GB/s interconnect bandwidth with 250-MHz clock frequency). In this work, we dedicated 4x more on-chip bandwidth than the configured PCIe link capability (4 GB/s per direction) to avoid the bottleneck at the on-chip crossbars and to obtain the full potential of near-storage processing in multi-VM workloads. However, the proper data stream width may vary depending on the target operators and resource budgets.



(a) SSD read (b) SSD write

(c) FPGA encryption (d) FPGA decryption

Figure 15: SSD and FPGA operation performance with different virtualization mechanisms.

The FPGA operators have a broad spectrum of resource utilization and data-processing throughput based on their algorithm complexity and synthesis strategies. Table 3 shows the benchmarks and their FPGA implementation results. The Blowfish data encryption and decryption operators consume a large number of registers, but demonstrate the lowest data processing performance among the benchmarks. On the other hand, the aggregate, K-nearest neighbors (KNN), and bitmap operators show the highest ideal performance due to their algorithmic simplicity.

5.2 Device Virtualization Performance

In this experiment, we measured the performance of the hardware-assisted virtualization mechanism and compared it with the native device performance. For this evaluation, we installed the SSD and FPGA devices through PCIe and implemented the Blowfish encryption and decryption operators [28] on the FPGA. To fairly compare the performance for both device types, we measured the performance of the computation and storage units separately. To measure the storage performance, we ran flexible I/O tester (FIO) [2] in both native and FlexCSV virtualization environments with an increasing data block size. In contrast, we leveraged the NVMe-extended protocol to measure the FPGA virtualization performance without invoking storage operations. Similar to the storage benchmarks, the FPGA benchmarks measured the end-to-end performance in both native and FlexCSV environments with an increasing data block size.

The experimental results show that our FCSV-Engine prototype delivers near-native NVMe SSD and FPGA performance.

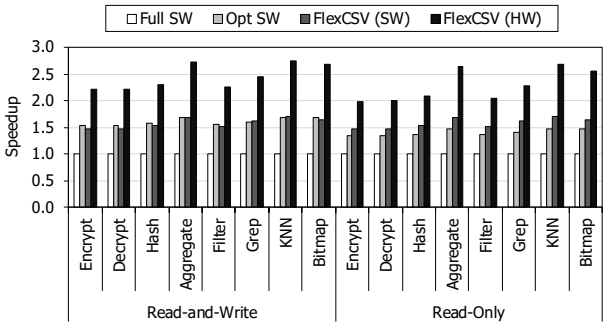


Figure 16: Speedup comparison in various virtualized environments.

Figure 15 shows the native and virtualized SSD and FPGA acceleration performance. Due to the hardware-assisted virtualization mechanism (including SR-IOV), FCSV-Engine can achieve the near-native performance when utilizing both SSD and FPGA devices in virtualized environments. Moreover, as the block size increases, the native and virtualized executions of the SSD and FPGA operators demonstrate the ideal bare-metal throughput (explained in Section 5.1 and Table 3). The increasing data block size further mitigates the software overhead by merging multiple near-storage processing requests using NVMe's scatter-gather list (SGL) support.

5.3 Near-Storage Processing Performance

In this experiment, we measured the performance of the near-storage processing benchmarks through existing software stacks and FlexCSV virtualization mechanisms. For this evaluation, we generated a 4-GB dataset as an input file of near-storage processing, and each benchmark running on a guest OS divides the dataset into multiple 4-KB blocks and iterates them to cover the total dataset size. To compare the speedup values over the existing software-centric mechanisms, we also executed the same near-storage processing benchmarks on the paravirtualization schemes with and without SR-IOV support at the SSD side (described in Figure 4). In addition, we measured the performance of software FCSV-Engine (described in Figure 12) to highlight the benefits of FlexCSV's hardware-level resource orchestration.

Each near-storage processing benchmark listed in Table 3 is executed in two VM workload scenarios. The first scenario reads a data block from the SSD directly, performs data processing using the FPGA's stream operator, and writes its output data to the SSD (read-and-write). The other scenario, on the other hand, follows the same routines for every data block, but does not write back the output data for further near-storage processing (read-only).

The experimental results show that FlexCSV can achieve 2.1x (geomean) faster near-storage processing in virtualized environments over the software virtualization mechanisms. Figure 16 shows the speedup values compared to the full software virtualization mechanism. In contrast to the software-based virtualization mechanisms, FlexCSV allows the VM to access the underlying devices and orchestrates the data movements between the virtualized units without any software intervention. The encryption and decryption benchmarks obtain 2.2x speedup (33 MB/s→72 MB/s) in the read-and-write case and 2x speedup (48 MB/s→95 MB/s) in the read-only case. The KNN benchmark, which shows the highest speedup values among the benchmarks, achieves 2.8x speedup over the full software virtualization (38 MB/s→105MB/s in the read-and-write case). Compared to the software FCSV-Engine implementation, the hardware FCSV-Engine implementation achieves 1.4x speedup on average.

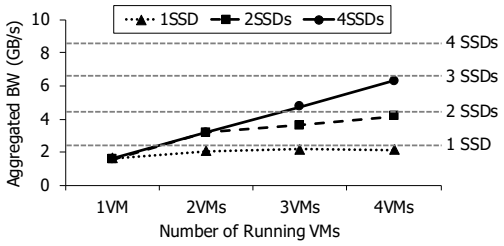


Figure 17: The aggregated bandwidth with a different number of VMs and SSDs.

5.4 Multi-SSD Performance

In this work, we evaluated FlexCSV's decoupled architecture by executing an I/O-intensive VM workload with an increasing number of VMs and PCIe-attached SSDs. The result is shown in Figure 17. In this experiment, the VMs utilize about 2 GB/s storage bandwidth, similar to the maximum bandwidth of a single SSD. A single SSD can meet the performance requirements of a single VM workload as there is no storage interference. However, when we utilize a single SSD and run two or more VMs, the aggregate bandwidth of all the VMs is limited by a single SSD. So, the tightly-coupled architectures suffer from such bandwidth imbalances and unexpected delays as the available storage bandwidth cannot be scaled easily. However, with the FPGA and SSD decoupled, the required bandwidth of multiple VMs can be met by adding more SSDs. In this work, FCSV-Engine supports attaching up to four SSDs to a single FPGA and achieves the scalable performance with the increasing number of VMs and SSDs.

5.5 Dynamic Resource Scheduling

In this experiment, we evaluated the effectiveness of FCSV-Engine's dynamic resource allocation and scheduling mechanisms. For this evaluation, we ran four VMs concurrently and generated the VM workload by following Poisson distribution with diverse expected request rates ($\lambda = \frac{1}{T} = \frac{1}{T_{exec}+T_{wait}}$). We ran four VMs and grouped the VMs into two groups (A, B) that have different request rates (T_{wait}/T_{exec}). After that,

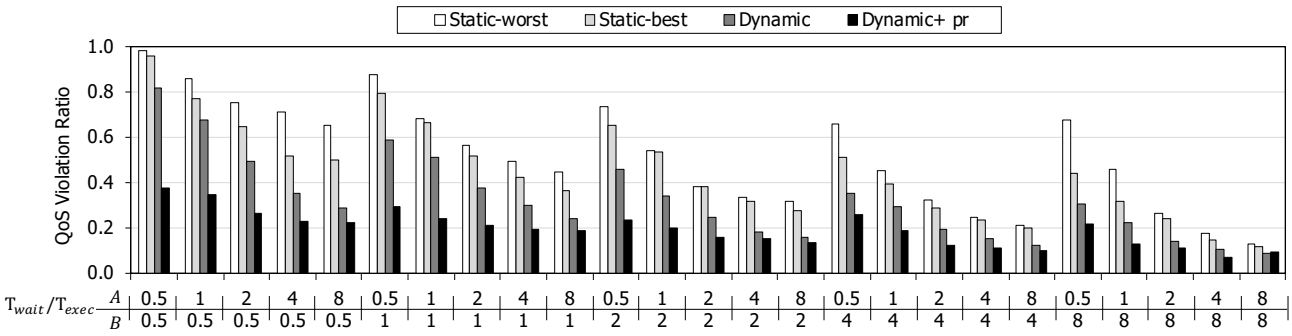


Figure 18: QoS violation ratios with different request rates from multiple VMs.

we compared their QoS violation ratios with four different resource scheduling strategies: (1) static-worst, (2) static-best, (3) dynamic, and (4) dynamic + partial reconfiguration (dynamic+pr). The static-worst, static-best, and dynamic scheduling methods utilize two physical operators, and dynamic+pr can support up to four physical operators through partial reconfiguration. The VMs in the experiment perform an encryption operation with near-storage processing capabilities.

Figure 18 shows the QoS violation ratios ($> 1.5 \times T_{exec}$) with diverse request rate combinations of the two VM groups. Except when the request rate between the two groups is the same, the static-best scheduling methods achieve lower the QoS violation ratios compared to the static-worst method. Also, the dynamic scheduling significantly lowers the QoS violation ratios over the static-worst and static-best strategies for all the workload scenarios. When both VM groups invoke the operators frequently (e.g., [A=0.5, B=0.5]), FCSV-Engine can increase the number of available operators using its partial reconfiguration support and further reduce the QoS violation ratios. If the request rate is low (e.g., [A=4, B=8]), the dynamic scheduling can drop QoS violations, but increasing the number of operators has a minor impact.

6 Related Work

Near-storage processing. Recent near-storage processing studies have proposed flexible FPGA overlay architectures and improved the usability of computational storage devices [33, 35]. Their FPGA overlay architectures can implement user-specified operators and activate them selectively through the crossbar switches and control logic. Also, their custom software stacks provide abstraction layers to hide the complexity of the underlying FPGA implementations.

NVMe SSD virtualization. NVMe virtualization becomes one of the most critical components in cloud environments to meet the performance demand from modern server workloads. For example, Amazon Web Services (AWS) accelerates I/O virtualization through dedicated hardware components. To make full use of its parallel and high-performance storage protocol, storage performance development kit (SPDK) vhost-nvme extends the SPDK library to provide virtual NVMe controllers to QEMU-based VMs [38]. Similarly, another implementation provides a mediated passthrough mechanism in kernel space with an active polling mode [29]. In addition, current hardware-assisted virtualization studies demonstrate offloading NVMe virtualization stacks to a programmable FPGA or SmartNIC device [22, 25].

FPGA virtualization. The existing FPGA virtualization studies introduce abstractions for FPGA logic cell and interconnection components [19, 20, 40]. First, user logic is encapsulated in flexible operators to be dynamically scaled and remapped to the physical fabric. Second, the host software manages the mapping between operators and physical FPGAs. To enable flexible mapping, high-level operators encapsulate information to enable the host software to generate new FPGA implementations on demand. Moreover, the host software maintains a registry to hide the latency of partial reconfiguration for dynamic scalability.

7 Conclusion

In this work, we propose a fast, flexible, and cost-effective mechanism to virtualize computational storage devices. The key idea is to use FCSV-Engine, an FPGA card designed to maximize the performance and cost-effectiveness of computational storage virtualization. It achieves high virtualization performance by applying hardware-assisted virtualization and resource orchestration. In addition, it achieves high cost-effectiveness by dynamically constructing many virtual computational storage devices and scheduling their near-storage processing at the hardware level.

Acknowledgments

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1901-12. We also appreciate the support from Automation and Systems Research Institute (ASRI) and Inter-university Semiconductor Research Center (ISRC) at Seoul National University.

References

[1] Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

[2] Flexible I/O Tester. https://github.com/axboe/fio.

[3] Intel Solid State Drives. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives.html.

[4] Linux KVM. https://www.linux-kvm.org/page/Main_Page.

[5] QEMU. https://www.qemu.org/.

[6] Samsung SmartSSD Computational Storage. https://samsungsemiconductor-us.com/smartssd/.

[7] Xilinx Alveo U250 FPGA. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html.

[8] Xilinx FPGAs & 3D ICs. https://www.xilinx.com/products/silicon-devices/fpga.html.

[9] Xilinx QDMA Subsystem for PCI Express. https://www.xilinx.com/products/intellectual-property/pcie-qdma.html.

[10] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs. In *2017 USENIX Annual Technical Conference (ATC 17)*, pages 167–179, 2017.

[11] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990, 2020.

[12] 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. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[13] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.

[14] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale DNN processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2018.

[15] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 153–165. IEEE, 2016.

[16] Ajay Gulati, Irfan Ahmad, Carl A Waldspurger, et al. Parda: Proportional allocation of resources for distributed storage access. In *FAST*, volume 9, pages 85–98, 2009.

[17] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, et al. Bluedbm: An appliance for big data analytics. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13. IEEE, 2015.

[18] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42. IEEE, 2018.

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

[20] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on FPGAs? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 991–1010, 2020.

[21] Dongup Kwon, Jaehyung Ahn, Dongju Chae, Mohammadamin Ajdari, Jaewon Lee, Suheon Bae, Youngsok Kim, and Jangwoo Kim. DCS-ctrl: a fast and flexible device-control mechanism for device-centric server architecture. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 491–504. IEEE, 2018.

[22] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo Kim. FVM: FPGA-assisted Virtual Device

Emulation for Fast, Scalable, and Flexible Storage Virtualization. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 955–971, 2020.

[23] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. SplitX: Split Guest/Hypervisor Execution on Multi-Core. In *WIOV*, 2011.

[24] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang Seok Ki. SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD. *IEEE Computer Architecture Letters*, 19(2):110–113, 2020.

[25] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK Ports, Irene Zhang, Ricardo Bianchini, Haryadi S Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 591–605, 2020.

[26] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. A hypervisor for shared-memory fpga platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 827–844, 2020.

[27] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia De Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-mei Hwu. DeepStore: in-storage acceleration for intelligent queries. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 224–238, 2019.

[28] Shraphalya B Nalawade and Dhanashri H Gawali. Design and implementation of blowfish algorithm using reconfigurable platform. In *2017 International Conference on Recent Innovations in Signal processing and Embedded Systems (RISE)*, pages 479–484. IEEE, 2017.

[29] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. MDev-NVMe: A NVMe storage virtualization solution with mediated pass-through. In *2018 USENIX Annual Technical Conference (ATC 18)*, pages 665–676, 2018.

[30] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, 2014.

[31] 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. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014.

[32] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases*, pages 62–73. Citeseer, 1998.

[33] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *2019 USENIX Annual Technical Conference (ATC 19)*, pages 379–394, 2019.

[34] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[35] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. Accessible Near-Storage Computing with FPGAs. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–12, 2020.

[36] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. Flin: Enabling fairness and enhancing performance in modern nvme solid state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 397–410. IEEE, 2018.

[37] Kurt K Ting, Steve CL Yuen, Kin-Hong Lee, and Philip HW Leong. An fpga based sha-256 processor. In *International Conference on Field Programmable Logic and Applications*, pages 577–585. Springer, 2002.

[38] Ziye Yang, Changpeng Liu, Yanbo Zhou, Xiaodong Liu, and Gang Cao. SPDK Vhost-NVMe: Accelerating I/Os in Virtual Machines on NVMe SSDs via User Space Vhost Target. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 67–76. IEEE, 2018.

[39] Joohyeong Yoon, Won Seob Jeong, and Won Woo Ro. Check-in: in-storage checkpointing for key-value store system leveraging flash-based SSDs. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 693–706. IEEE, 2020.

[40] Yue Zha and Jing Li. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 845–858, 2020.

[41] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, et al. FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 477–492, 2018.