Reconfigurable Virtual Memory for FPGA-Driven I/O

Joshua Landgraf
The University of Texas at Austin
Austin, Texas, USA
jland@cs.utexas.edu

Matthew Giordano
The University of Texas at Austin
Austin, Texas, USA
mgiordano@utexas.edu

Esther Yoon
The University of Texas at Austin
Austin, Texas, USA
estheryoon@utexas.edu

Christopher J. Rossbach
The University of Texas at Austin
Katana Graph
Austin, Texas, USA
rossbach@cs.utexas.edu

ABSTRACT

FPGAs are increasingly used to accelerate modern applications, and cloud providers offer FPGA platforms on-demand with a variety of FPGAs, I/O peripherals, and memory options. FPGA vendors expose I/O with low-level interfaces that limit application portability. Current approaches to abstracting these interfaces trade level of abstraction against performance.

We present FSRF, File System for Reconfigurable Fabrics, which abstracts FPGA I/O at a high level without sacrificing performance. Rather than exposing platform-specific I/O interfaces, FSRF enables files to be mapped directly into FPGA virtual memory from the host. On the FPGA, powerful OS-managed virtual memory hardware provides performant access to FPGA-local resources and helps coordinate access to remote data. FSRF leverages reconfigurability to specialize its virtual memory *implementation* to applications, including selecting between SRAM and DRAM TLBs, adapting FPGA DRAM striping, and tuning DMA I/O. On Amazon F1 FPGAs, FSRF outperforms techniques from FPGA OSes such as Coyote and AmorphOS with improvements of up to 64× and 2.3×, respectively (+75% and +27% geometric mean), and performance close to that of physical addressing (90% geometric mean).

CCS CONCEPTS

• **Software and its engineering** → **Virtual memory**; • **Hardware** → *Reconfigurable logic and FPGAs.*

KEYWORDS

Virtual Memory, FPGAs, Operating Systems, Virtualization

ACM Reference Format:
Joshua Landgraf, Matthew Giordano, Esther Yoon, and Christopher J. Rossbach. 2023. Reconfigurable Virtual Memory for FPGA-Driven I/O. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3582016.3582048

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582048

1 INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) are reconfigurable accelerators, providing hardware-like efficiency with software-like customizability. Developers use FPGAs to create datapaths specific to their application, maximizing the utilization of functional units, on-chip cache, and off-chip memory. This enables them to outperform CPUs by orders of magnitude [25, 92] without the costs and limitations of ASIC development.

FPGAs' tight coupling of memory and logic makes them competitive for a wide variety of workloads, including communication [23, 51, 55, 92, 108], databases [24, 57, 74], image processing [81], finance [52, 69], graph processing [33, 82], and machine learning [29, 97, 104, 121]. Their growing capacity and capabilities have led to their adoption by cloud providers such as Amazon [37], Microsoft [79], Alibaba [4], Baidu [13], Huawei [48], and Tencent [107]. However, with increasing demand for, and availability of, FPGAs comes the need for simplified development workflows and greater application portability. This requires abstraction of not just FPGA fabric, but also external FPGA resources as well.

While many previous FPGA operating systems have focused on how to abstract on-chip fabric [60, 66, 78, 118], abstracting FPGA I/O remains a significant challenge. An FPGA OS, ideally provides critical features and properties (§3) including efficiency, compatibility, portable I/O interfaces, and automated management of the storage and memory hierarchy. Existing approaches forfeit at least one of these goals to meet the others. Some FPGA OSes [60, 78] simply multiplex available I/O resources, often in a coarse-grained fashion. This approach can have low hardware overheads, but fails to provide portable high-level interfaces that are useful to developers, forcing them to implement data retrieval and storage stacks for each application. In contrast, FPGA OSes that heavily abstract the communication process [1, 30, 68, 96, 99, 118] can place a considerable burden on system, compiler, and runtime developers. This, in turn, can lead to poor performance that application developers have little control over. Even state-of-the-art systems [66] can neglect important classes of workloads with design decisions such as using SRAM TLBs, which struggle to scale to larger working sets. While abstraction is expected to come at some cost, high overheads can disincentivize system adoption, especially when performance and efficiency gains motivate the use of FPGAs.

We argue that a balanced and practical approach to FPGA I/O abstraction can provide portability, compatibility, and performance

without such compromises. Reconfigurability is key to making this possible. We demonstrate this with FSRF (File System for Reconfigurable Fabrics), a new solution for FPGA I/O. FSRF makes data accessible through an `mmap`-like file system interface and automates data movement throughout the storage and memory stack in response to application needs. Previous systems have supported mmap techniques for accelerators accessing host data [90, 98]; FSRF uses similar techniques but specializes the design for FPGAs and specializes *implementation* for applications using reconfiguration. Data is accessed via a virtualized version of the industry-standard AXI4 [10] interface, enabling compatibility with a large variety of applications without impacting their portability between different hardware platforms. FSRF accomplishes this via a new, FPGA-optimized virtual memory system that selects between using FPGA-local DRAM or on-chip SRAM for caching translations, enabling fast access to large working sets, even for applications with random access patterns. A key insight is that reconfigurability can be used to tailor FSRF's implementation to the task at hand.

We implement FSRFin the Amazon F1 cloud and evaluate it with a variety of demanding workloads, finding that it can maximize the throughput of modern FPGA hardware while greatly simplifying the process of accessing data for a diverse set of applications. This is accomplished without exposing the particular peripherals or architecture of the target hardware, and without requiring significant changes to existing applications. We show this functionality has minimal overhead (90% of physical addressing) is up to 64× faster (75% overall) vs. the state of the art, Coyote, and performs up to 2.3× faster (+27%) vs. AmorphOS. We make the following contributions:

- We describe a design for an `mmap`-based virtual memory system for FPGAs that provides performant access to file-backed data without exposing hardware specifics;
- FSRF leverages reconfigurability to specialize the implementation of its virtual memory hardware to maximize per-application performance;
- FSRF transparently moves data throughout the memory hierarchy, enabling oversubscription of FPGA-local memory (a first for FPGA OSes);
- FSRF modulates between DRAM- and SRAM-based TLBs to save area while enabling local address translation to cover *all of memory, even without huge pages.*

2 BACKGROUND

Field Programmable Gate Arrays (FPGAs) are circuits that can be (re-)configured to implement custom logic. They can be deployed in many ways, including: standalone (e.g. network switches [5]), in SoCs [6, 32], on I/O pipelines [106], or I/O-attached (e.g. via PCIe) to offload compute. We focus on offload configurations as they are the most common case for on-demand cloud FPGAs [37, 79].

Currently, on-demand FPGAs in the cloud [37], only support coarse-grain sharing. F1 supports SDKs for developing hardware accelerators, saved as Amazon FPGA Images (AFIs), which update the entirety of the FPGA fabric when deployed. Abundant recent work on FPGA sharing [60, 66, 78, 109, 118, 119], has not yet impacted production settings. Previous FPGA OS proposals explored cross-application sharing [28, 50, 112], hardware abstraction layers [47, 61, 62, 77, 80, 114], shared runtime support [35, 44, 103], and access from a virtual machine [78, 87]. Theoretical aspects of FPGA scheduling [28, 40, 102, 111], heterogeneous scheduling [7, 20, 43, 102, 111], preemption [72], relocation [53], and context switch [70, 94] are well-explored. While prior work has explored FPGA access to OS-managed resources such as virtual memory [1, 30, 75, 117] and file systems [100], there remains an urgent need for solutions that are both portable and efficient, which is our focus.

Recent designs for FPGA sharing in datacenters [22, 27, 38, 60, 64, 66, 113, 118, 119] use reconfiguration or partial reconfiguration to share fixed partitions of FPGA fabric among applications with bitstreams pre-compiled to target those partitions. While there are variations in this space we focus on supporting performant I/O for a design of this form, which we assume uses some form of address translation to enforce cross-domain protection/isolation.

The current states-of-the-art in OS FPGA support are AmorphOS [60] and Coyote [66], both of which support multi-tenancy. AmorphOS extends processes with an abstraction for FPGA-based execution and can spatially share an FPGA among applications. Coyote [66] is a shell for FPGAs which supports both spatial and temporal multiplexing as well as communication and virtual memory management. While Coyote provides interfaces for networking and memory, it does so using a fixed mechanism for address translation (software-managed, configurable-size SRAM TLBs) and it does not support oversubscription. FSRF extends a design such as AmorphOS or Coyote with performant and flexible access to I/O through the file system.

3 MOTIVATION

In the wake of Moore's Law, hardware specialization is the *de facto* approach to improving performance, scalability and energy efficiency [45]. Creating new hardware is slow and expensive, and provisioning shared infrastructure with diverse fixed function accelerators is untenable [93]. FPGAs deployed as arbitrarily reconfigurable *universal* accelerators, can enable infrastructure providers to meet diverse specialization needs while retaining the simplicity of single-SKU provisioning. However, to enable this vision software layers that support protected sharing, virtualization, compatibility, portability, and *access to other OS-managed resources* are necessary.

Research exploring protected FPGA sharing [26, 44, 50, 75, 84, 99, 100] based on static partitioning [22, 27, 38, 63, 64, 78, 113] and dynamic partitioning [60, 66, 68, 118] has yielded techniques for sharing on-fabric resources by *mediating access* to OS-managed resources, but has largely retained traditional abstractions for those resources and neglected the opportunity to leverage reconfigurability to specialize the implementation of those abstractions to individual application needs. We argue that *compatibility* requires better access to virtual memory and the file system, particularly as consolidated cloud settings can often be subject to oversubscription that is unpredictable by a developer at design time.

Improved techniques for FPGA virtual memory are also necessary because *FPGA applications do not access memory like CPUs or GPUs do.* In CPUs and GPUs, caches automatically manage the process of coalescing small memory accesses into larger memory transactions, and handle the process of prefetching data that will

be needed in the near future. In FPGAs, register sizes, cache architecture, and prefetching are largely implemented by the user to suit the needs of applications. Applications access memory through the same bus protocols they use to access peripherals. Applications streaming large amounts of data can request entire pages at a time, while applications performing random accesses can make many smaller requests. Furthermore, many FPGA memory systems decouple read requests from the return of read data, and write requests from the return of a write persistence response, enabling prefetching of data far ahead of its use.

These key differences mean FPGAs do not always benefit from a fixed CPU-like memory hierarchy design (e.g. on-chip TLBs). While the smaller accesses of CPUs and GPUs will often hit the same page (and corresponding TLB entry) repeatedly, FPGA applications often tune their access sizes to request as much data from a page as they need at that time, meaning that the corresponding TLB entry will likely not be reused. FPGA applications often prefetch data far enough in advance to hide the latency of the memory entirely. We therefore find it worthwhile to explore new approaches to address translation and memory hierarchy design for FPGAs that can take advantage of and support the unique needs of FPGA applications.

Limitations in State-of-the-Art. Current state-of-the-art FPGA systems struggle to balance the need to deliver performant I/O while also providing sufficient abstraction for usability. For instance, Cascade [96] and Synergy [68] completely abstract away I/O interfaces through the use of "unsynthesizable" Verilog system tasks. Verilog logic is converted into a state machine that can pause program execution at sub-clock-tick granularity to create the illusion that the I/O is performed instantaneously. While this provides a familiar `libc`-like programming model with APIs like `fopen` and `fread`, it presents significant challenges for the runtime to overlap I/O with execution, resulting in poor performance.

In contrast, systems such as Coyote [66] and AmorphOS [60] expose the underlying hardware interfaces of their platform. Coyote uses a user-configurable TLB to enable virtual addressing of host, network, and local memory interfaces, while AmorphOS uses segment-based address translation. Both approaches burden developers with managing data placement, TLB sizing, and implementing system services, and require a large number of I/O signals to be routed to applications, regardless of whether they need them. Neither system directly supports the file system. FPGA OS support that delivers both performance and portability is urgently needed.

4 DESIGN

FSRF extends a generic FPGA OS design, such as AmorphOS or Coyote, to provide `mmap`-style virtual memory on FPGAs. On the host side, FSRF provides applications with functions for managing an accelerator's file descriptors and virtual memory mappings (§ 4.3). Data can then be accessed from the FPGA through a latency-insensitive, virtually-addressed AXI4 memory interface (§ 4.4).

An overview of FSRF is shown in Figure 1. Applications on the host use a FSRF-provided library (§ 4.3) to communicate with a FSRF daemon (§ 5.1) to open files, `mmap` them into FPGA virtual memory, and control FPGA applications. When applications access virtual address, translation is handled by a hardware memory management unit (§ 5.2). FPGA accesses to not-present memory are handled by

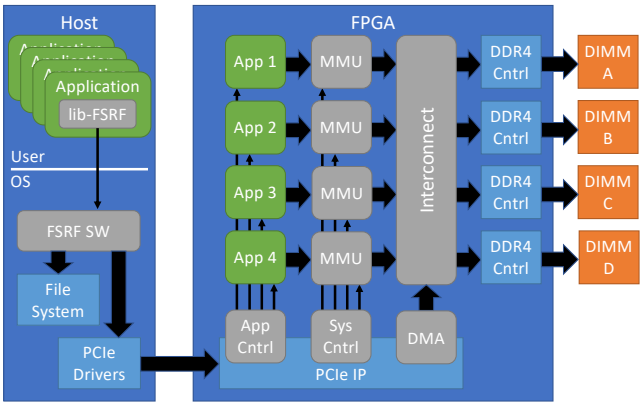

Figure 1: Overview of FSRF's design and implementation.

the daemon, which manages an FPGA-local page cache through communication with the host file system, DMAs between the host and FPGA (§ 5.4), and control messages to the MMUs. Translated accesses on the FPGA are then forwarded to the interconnect (§ 5.3) to access file data from FPGA DRAM.

4.1 File Abstraction for I/O Interfaces

FSRF integrates with the host file system to provide a single, abstract point of access to data from a variety of sources including stable storage, networking, and virtual services. This design choice is driven by increasing connectivity of modern FPGAs and the need for scalability for system software. FPGAs can be equipped with on-board networking and storage, and can access data from external NICs and SSDs via the PCIe bus. They can feature multiple types of memory, such as local DRAM, GDDR, and HBM; remote DRAM via an interconnect (e.g. Intel QPI); and storage-class memory (e.g. 3D XPoint). Without abstraction, application developers face a growing variety of peripherals, each of which can require unique interface implementations. FPGA OSes, such as AmorphOS [60] and Coyote [66], multiplex on-device I/O resources and present each application with a unique port for each interface. Under this scheme, routing connections from N applications to M interfaces can require $N \times M$ interconnections for maximal bandwidth. Providing a single file-based I/O abstraction reduces developer effort and makes accelerators more portable as they no longer need to implement interface-specific functionality.

4.2 Virtual Memory Abstraction and `MMAP`

To make data accessible from the FPGA, FSRF uses memory-mapping. An `mmap`-like interface on the host is used to convert file-based data locations into virtual addresses that can referenced on the FPGA. While lower-level than the `libc`-like abstractions supported by Borph [100], Cascade [96] and Synergy [68], virtual addressing avoids the high costs associated with forwarding accesses to the CPU by enabling most I/O accesses to be handled entirely on-chip. Memory mapping provides the best balance of functionality, abstraction, and performance because it divides the labor of accessing data between host CPU and FPGA according their strengths.

```
1: package FSRF_SRAM_TLB;
2:   parameter NUM_TLB = 3;
3:   parameter PAGE_BITS[] = {12,21,32};
4:   parameter LOG_DEPTH[] = {10,6,2};
5:   parameter ASSOC[] = {2,1,2};
6: endpackage
```

Figure 2: Example of FSRF's compilation controls.

In FSRF, the host CPU maintains memory-mapping metadata, including tracking which resources back virtual addresses. This involves small, complex, and latency-sensitive operations that CPUs handle well. Synchronization and coherency are software-initiated to reduce unnecessary communication overheads. FPGA MMUs and DMA engines are also software-managed to enable the hardware to support a wide range of policies efficiently. We take this one step further on FPGAs with their own DRAM by caching data and offloading virtual memory accesses to FPGA logic. This eliminates system call and software overheads in the common case, enabling the accelerator to operate at near-native performance.

FSRF differs from prior work in two other key ways. The first is having files back virtual memory addresses. Prior work, such as Coyote [66] and Optimus [78], use host memory to back virtual addresses. This requires them to pin host pages to make them accessible from the FPGA, preventing data movement throughout the memory hierarchy. With FSRF, data can be prefetched, faulted in, and migrated back to storage without interrupting accelerator execution, providing the greatest possible performance and flexibility.

FSRF also decouples accelerator virtual memory from host virtual memory. Current FPGA systems focus on supporting the 4 KiB and 2 MiB page sizes used by modern x86 CPUs. However, increasingly popular alternatives, such as ARM and RISC V CPUs, do not necessarily use these page sizes. For instance, the ARM A15 uses 64 KiB, 4 MiB, and 16 MiB huge pages [9], while the ARM A9 in Xilinx SoCs does not support huge pages at all. Furthermore, CPUs like Apple's M1 use 16 KiB base pages by default [49]. Decoupling these virtual memory systems enables FSRF to retain control over its page sizes for better performance and portability.

4.3 Host Interface

FSRF provides two host-side interfaces for managing its operation: a compile-time interface for tweaking hardware implementation and a run-time interface for managing file mappings. The compile-time interface enables developers to optionally adjust FSRF's hardware configuration based on application needs. While exact options available will depend on the backend platform, Figure 2 demonstrates the configuration for the SRAM TLB from our evaluation. The parameters shown control the number of TLBs and (in powers of two) their page sizes, capacities, and associativities. This enables developers to trade off FPGA area for functionality, with FSRF's host software adapting to the changes accordingly.

File mappings are managed using the API in Figure 3. Files can be opened and closed (`fsrf_open`, `fsrf_close`) and mapped to / unmapped from FPGA virtual memory (`fsrf_mmap`, `fsrf_munmap`) with POSIX-style semantics. Data synchronization can

```
 1: #define FSRF_ADV_STREAM 1
 2: #define FSRF_ADV_RANDOM 2
 3: #define FSRF_ADV_REUSE  4
 4: #define FSRF_ADV_LOWUSE 8
 5: fsrf_rc fsrf_open(int* fd,
 6:   char* path, int prot, int mode);
 7: fsrf_rc fsrf_close(int fd);
 8: fsrf_rc fsrf_mmap(void** addr,
 9:   size_t len, int prot, int flags,
10:   int fd, size_t offset);
11: fsrf_rc fsrf_munmap(void* addr,
12:   size_t len);
13: fsrf_rc fsrf_msync(void* addr,
14:   size_t len, int flags);
15: fsrf_rc fsrf_madvise(void* addr,
16:   size_t len, int advice);
```

Figure 3: FSRF's C API for FPGA file mappings.

be triggered both to and from the host with `fsrf_msync` via flags. Finally, access pattern metadata can be provided via `fsrf_madvise`. Flags, such as `FSRF_ADV_STREAM` and `FSRF_ADV_RANDOM`, hint at locality, while flags like `FSRF_ADV_REUSE` and `FSRF_ADV_LOWUSE` hint at data cacheability. These help FSRF make the best policy decisions for the target workload, including which virtual memory hardware and policies to use.

4.4 FPGA Interface

In previous FPGA OSes, applications manually stream data from the host and manage caching in device-side memory. Instead, FSRF automatically handles paging and caching, so these interfaces are not necessary. FSRF presents applications with a virtually-addressed AXI4 memory interface [10] for data access. AXI4 can efficiently handle a wide variety of access patterns, works with IP from many vendors, supports high-level synthesis code (e.g. Xilinx Vitis HLS), and is interoperable with other standards like Intel's Avalon bus.

AXI4 includes 5 independent channels: two for initiating R/W transactions, two for R/W data, and one for write completion metadata. Transaction channels specify the starting address, the number of words, and an ID for ordering. Data channels carry both data and metadata, such as the transaction ID and an error response. The write response channel signals write completions, including the corresponding ID and write response.

To access data, applications generate AXI4 transactions to the corresponding virtual addresses. FSRF handles transaction buffering so requests and data can be pipelined for maximal throughput.

4.5 Reconfigurable TLBs

Prior work in FPGA virtual memory has often mirrored that of CPU designs, using segments [60] or small SRAM TLBs [66] for fast address translation on-chip. While these mechanisms are sufficient for many workloads, their limited capacity ultimately requires systems to trade-off TLB reach with granularity of memory management, leading to poor performance in important classes of algorithms and datasets. FSRF takes the position that there is no one-size-fits-all

solution to address translation for to FPGA applications, and exploits the reconfigurability of FPGAs to tailor address translation mechanisms to application requirements for optimal performance.

For workloads that benefit from fine-grain management of large amounts of memory, FSRF offers a specialized address translation mechanism: a DRAM TLB which uses a portion of FPGA-local, off-chip memory for caching TLB entries. This provides ample TLB capacity at the cost of increased translation latency and of a small amount of memory bandwidth and capacity. However, since FPGA workloads often heavily pipeline memory requests, the additional translation latency can often be hidden entirely. Likewise, the small amount of memory bandwidth and capacity the TLB uses is worthwhile as it overall improves performance by reducing misses that would have otherwise needed to be handled by the host.

FSRF also offers a more traditional SRAM TLB option. At compile time, the page sizes, number of TLB entries, and cache associativity can be configured as covered in Sec 4.3, enabling a wide range of possibilities for workloads that do not need particularly large TLBs.

4.6 Oversubscription

FSRF supports oversubscription of device-side memory using similar mechanisms to swapping-based virtual memory. When the device-side MMU cannot find data in local memory, it waits for outstanding I/O to complete and generates a page fault for the host to handle. The host then evicts pages (currently via FIFO policy) from device memory before loading the requested data on the device.

4.7 OS-Defined Striping

Striping splits contiguous data across multiple DIMMs, increasing memory parallelism. While this can increase performance when applications have exclusive access to memory, it can increase the chance of collisions and reduce performance when DIMMs are shared. To minimize these conflicts, FSRF either implements a coarse-grain isolated-sharing strategy (e.g. 1 application to 4 DIMMs, 2 applications to 2 DIMMs each) in hardware or sets virtual-to-physical page mappings to achieve a fine-grain solution from software. FSRF uses the latter in cases where there may be significant differences in application memory use, bandwidth, or access pattern, or when DIMMs cannot be evenly split among applications.

4.8 Stream Support

Some data sources (e.g. pipes and network streams) are not seekable and so require special consideration. Normally, these would be accessed using a dedicated streaming interface, such as AXI4-Stream, which provides ordering, packet delimitation, and low signal overheads. However, many of these signals are already present in the standard AXI4 interface.

Instead, FSRF provides two modes of operation: memory and streaming. In memory mode, streams are adapted to be fully compatible with AXI4 and mimic a memory-mapped FIFO. Reads and writes to the virtual stream address can attempt to block until the stream is ready, but can time out and return an error if the transaction cannot complete in a timely manner. Applications will not receive stream packet termination signals in this mode, but they will have more control over how stream data is interleaved with standard file data due to AXI semantics.

In streaming mode, an extra signal is provided on data channels for streaming operations. Applications create streams by making an AXI4 transaction to the virtual stream address, which sets the stream width and assigns it an ID. The streaming signal is then used to indicate incoming or outgoing stream data, with the AXI last signal used to delineate packet termination when supported. Steams can be closed in a similar manner to how they were opened, and if closed on the opposite end first, a decode error (invalid interconnect address) will be returned upon attempting to access it.

4.9 Contrast with Prior Work

AmorphOS implements minimal segment-based memory virtualization hardware, specializing in fabric- and interface-sharing techniques. AmorphOS requires host applications to push data to the FPGA in advance of its use, and supports segmentation as the only means to manage memory mappings. FSRF, on the other hand, can dynamically fault in data to natively overlap computation and communication, and uses configurable TLBs for much finer-grained control of memory management. These enable features, such as oversubscription, that AmorphOS lacks. Compared to AmorphOS' custom memory interface, which only supports 64-byte requests, FSRF uses AXI4, which can natively encode sequential accesses for better memory access efficiency. Finally, FSRF's flexible striping policy intelligently splits data across DIMMs when it is not detrimental to performance, eliminating DIMM collision issues that can arise from AmorphOS' striping scheme.

Coyote is the most similar prior work to FSRF, but it still reflects a number of different design decisions. While Coyote's approach to virtual memory only transfers data at page or request granularity, FSRF's approach can adapt both the page and transfer size to suit application needs. Through the use of a metadata interface, and by monitoring page faults at run time, FSRF is able to adapt these policies to maximize DMA bandwidth and FPGA memory utilization. FSRF's use of files for backing storage also enables explicit support for oversubscription, both of FPGA and host page cache memory, for better access to large datasets. In contrast to AmorphOS, Coyote is optimized for streaming memory workloads, and uses a custom interface for requests generated by the FPGA. FSRF's use of AXI4 enables comparable streaming performance, but with greater flexibility, compatibility, and transaction metadata for developers who desire such features. While Coyote only supports SRAM TLBs with a limited selection of page sizes, FSRF offers customizable page sizes and a DRAM-based TLB. These enable efficient random access to large working sets without sacrificing granularity of memory management to cater to workloads of all types. Finally, FSRF's flexible striping policy enables performance benefits over Coyote's approach (§ 6).

OPTIMUS specifically targets Intel's HARP platform, where the FPGA is socketed alongside a CPU and must access DRAM through it. OPTIMUS enables host memory to be accessed from the FPGA by pinning host pages and configuring IOMMU entries for efficient address translation across the UPI link. While this approach does provide shared virtual memory across the host and FPGA, it does not provide the ability to dynamically fault in data like FSRF does, so demand paging and transparent oversubscription are not available. Since OPTIMUS is not moving data between host and FPGA-local

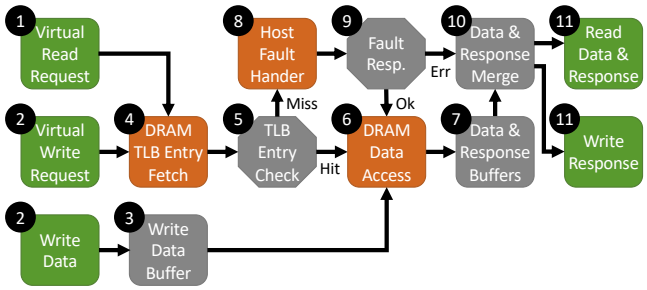

Figure 4: Control flow of FSRF's DRAM-backed TLB.

memory like FSRF, huge pages are always used to maximize the hardware-provided TLB's reach. Likewise, since memory accesses are always performed remotely, policies such as striping or batching are not supportable.

5 IMPLEMENTATION

We implement FSRF by extending AmorphOS since it is open source and runs on public cloud hardware (Amazon F1). We considered Coyote, which is also open source, but it is not currently available for cloud platforms.

5.1 FSRF Daemon

The FSRF daemon contains FSRF's host-side implementation. When an application opens and `mmap`s a file, a unique fault handler thread is created to interface with the corresponding MMU on the FPGA.

The fault handler polls the FPGA for TLB misses, dequeues them, and checks validity/permissions, returning errors when appropriate. If the requested data is not present in FPGA-local memory, the handler ensures there is enough free space by writing back and freeing resident pages when necessary. It then fetches the file data, DMAs it to FPGA DRAM, updates the TLB as needed, and sends a response to the MMU to continue.

FSRF's host component supports a number of policies for improving performance. Policies are chosen automatically unless explicitly set via an appropriate library function. A configurable I/O batch size (default 2 MiB) enables applications to trade off latency and throughput depending on their needs. FSRF also supports selecting between DRAM and SRAM TLB implementations when both have a corresponding bitstream compiled. For the SRAM TLB, FSRF uses 2 MiB pages when undersubscribed to populate memory as fast as possible and maximize throughput and TLB reach. Once oversubscribed, FSRF switches to a 4 KiB page and I/O size to minimize latency and thrashing. If data is accessed sequentially, FSRF progressively scales up the I/O size exponentially until either the stream ends (e.g. short bursts) or the maximum size of 2 MiB is reached. This enables efficient operation for both random and streaming workloads in both under- and over-subscribed conditions.

5.2 MMU with DRAM and SRAM TLBs

Virtual memory accesses are handled by FSRF's memory management unit (MMU), which uses a device-side TLB cache to translate accesses. FSRF supports TLBs in both FPGA-local DRAM and on-chip in SRAM, and can use control registers or reconfiguration to switch between them according to application needs.

DRAM TLB. When TLB entries are cached in FPGA-local DRAM, FSRF processes them in parallel via the pipeline summarized in Figure 4. The MMU receives accesses as virtually-addressed AXI4 read requests (1) and write requests combined with write data (2). Before processing write requests, FSRF waits until all their data has been buffered (3) to prevent their physical accesses from stalling after address translation.

Once a transaction is ready, the MMU uses its virtual address to index into a host-managed set-associative cache held in FPGA-local DRAM (4). This lookup returns a number of TLB entries based on the associativity of the cache. We currently configure FSRF's DRAM TLB with an associativity of 8 and a 4 KiB page size, resulting in a 48-bit virtual address space and a 36-bit physical address space. Accesses into DRAM are pipelined so that the TLB can process a translation every cycle provided sufficient memory bandwidth.

TLB entries are checked in parallel for validity, permission, and virtual address match (5). On a hit, the transaction is forwarded to the interconnect (6) using the physical address from the matching TLB entry. Read data and transaction responses from DRAM are buffered (7) until they are ready to be returned to the application.

If there are no hits, the MMU waits for outstanding memory accesses to complete and yields to the page fault handler on the host (8). The host provides a response (9) enabling the MMU to either proceed normally (6) or return an error. Data and responses from both the interconnect and the MMU are combined back into a single stream (10) and returned to the application (11).

SRAM TLB. FSRF also supports on-chip SRAM TLBs for applications with high locality and reuse. Since SRAM has very low access latencies, we manage translations entirely via a state machine, as opposed to the pipeline design of the DRAM TLB. The SRAM TLB is highly configurable and supports power of 2 page sizes (though FSRF's host software only supports 4K and 2M pages currently). Each supported page size has its own SRAM cache and hit evaluation logic, with tunable capacity and associativity. The SRAM TLB is otherwise quite similar to the DRAM TLB in how it accepts transactions, formats TLB entries, and handles TLB misses.

5.3 Interconnect

Once AXI transactions are translated to physical addresses, the interconnect routes them to the appropriate memory controller to access the corresponding DRAM DIMM. Generally, this would be implemented with a vendor-provided IP. However, our vendor's interconnect has a critical limitation: transactions issued with the same ID cannot be issued to different endpoints until the prior one completes, significantly limiting memory parallelism.

FSRF's interconnect takes a different approach. It issues transactions to memory controllers in-order and forces the controllers to process them in-order by assigning them the same ID downstream. This enables transactions to proceed to multiple memory controllers in parallel while ensuring they can be returned to applications in-order without deadlock, even when other applications are submitting transactions to the same DIMM.

FSRF's interconnect also features a custom arbiter for enhanced performance. The arbiter prioritizes PCIe transactions first for better throughput and latency. When there are no DMA requests, it accepts application requests according to a priority list, which is rotated every cycle to improve fairness and reduce starvation. The arbiter is pipelined to ease timing closure, and can accept new transactions every cycle for full throughput.

5.4 DMA

FSRF also provides its own DMA implementation. DMAs are often limited by handshake overheads, so FSRF simplifies communication to improve performance.

On the host, DMA buffers are allocated using 2 MiB huge pages to guarantee contiguity of physical memory. Since CPU bandwidth on the host is ample (70-80 GiB/s theoretical on F1), we are able to quickly coalesce file data into the DMA buffers. FSRF is then able to issue a single DMA command for the entirety of the buffer.

On the FPGA, FSRF includes its own DMA controller that takes advantage of the PCIe master (PCIM) interface on F1. This interface allows FSRF to generate its own PCIe DMAs, control buffering of data, and monitor their completion. FSRF's DMA controller includes a virtual queues for each FPGA application, allowing for concurrent requests from their respective fault handlers on the host and independent DMA completion tracking.

6 EVALUATION

We evaluate FSRF with the workloads summarized in Table 1. Our workloads use a variety of programming models (HDL and HLS) and cover several common uses cases for file and memory access patterns. These include purely streaming workloads (AES, CONV, FLOW, HLL, MD5, SHA, and SHA HLS) as well as workloads with data reuse and/or random access patterns (DNN, GUPS, NW, PGRNK, RNG, and TRI).

6.1 Workloads

AES is an AES-256 cryptography accelerator. Each instance uses 4 open-source AES cores [46] in parallel, enabling it to stream up to 32 GiB/s of data (16 GiB/s per read/write data channel) per instance.

CONV is an image convolution accelerator. It processes 64 1-channel pixels every cycle via a wide pipeline for a peak of 16 GiB/s throughput per read/write data channel.

DNN is an AI inference engine synthesized by the DNNWeaver [97] framework. We adapt its memory interface to AXI4 and modify its control API to facilitate running kernels back-to-back. DNN exhibits aspects of both sequential and random access patterns.

FLOW computes the optical flow for a set of images based on code from the Rosetta suite [123]. We widen the original datapath and streaming memory interface to be able to saturate the available bandwidth on F1.

GUPS is a random-access microbenchmark. For a number of 8-byte words, N, it performs $4N$ RMW updates to an $8N$-byte region of memory using a simple PRNG to determine the access pattern.

HLL is an accelerator for the HyperLogLog algorithm modified from [67]. It streams input data at full-throughput (16 GiB/s), and approximates the cardinality of the the target set of unique numbers.

MD5, *SHA*, and *SHA HLS* are streaming-style hashing accelerators that stream data at full-throughput (16 GiB/s), hash data in parallel, and accumulate the results locally. Operations are implemented using open-source MD5 [71] and SHA-256 [91] cores. SHA HLS uses Vitis HLS.

NW is a Needleman-Wunsch DNA sequence alignment accelerator. It uses a pipelined grid to compare two 128-bit (64 base-pair) sequences every cycle. NW iterates over all combinations of grid-aligned sub-sequences from two input strings and writes out alignment scores for a peak total bandwidth of roughly 4 GiB/s.

PGRNK is an iterative PageRank graph accelerator. It streams in vertices and incoming edges, fetches weights, and computes the updated pageranks. We use matrices from the SuiteSparse Matrix collection [34], including a small web connectivity matrix (webbase-1M [116]), an internet traffic archive (mawi_201512020030 [8]), and a large synthetic matrix (GAP-kron [16]).

RNG is a microbenchmark that generates random accesses using a PRNG with a period of 2^{19}. The core has a configurable access size, from 64 B to 2 MiB, in powers of two, and can generate new accesses every cycle.

TRI is a graph triangle counting accelerator. It uses a domain-specific data format designed to reduce neighbor lookup latency. Graphs are randomly generated with an average degree of 20 and roughly 1.3k triangles.

6.2 Methodology

We measure the execution of four concurrent workloads, each processing small (~32 MiB), large (~2 GiB), and oversubscribed (~32 GiB) datasets (collectively using 2× FPGA DRAM). Since TRI and GUPS have extremely long run times, we use their large datasets for their oversubscription experiments, and instead limit applications to 1.920 MiB (2 GiB - 128 MiB) of usable FPGA DRAM.

Each workload is measured with *cold*, *warm*, and *hot* data. *Cold* data is not initially cached, and must be loaded from an NVMe SSD. *Warm* data is initially cached in host DRAM, but not FPGA DRAM. *Hot* data is cached entirely in FPGA DRAM with corresponding TLB entries prepopulated as well, modulo limitations in TLB size. Since oversubscribed datasets exceed the size of FPGA DRAM, we only evaluate them with cold and warm data. Workloads are run on AWS f1.4xlarge instances to ensure the host has sufficient RAM for our warm, oversubscribed dataset experiments and for consistency with this case for our other configurations.

We report end-to-end runtime and system throughput (total bytes accessed divided by average application runtime). These metrics can differ when some applications finish faster than others, which can happen due to slight performance differences between F1's memory interfaces. Results are from an average of five runs.

We evaluate each workload using FSRF, Coyote and AmorphOS address translation, and physical addressing.

FSRF is configured to use either a DRAM- or SRAM-based TLB as indicated in Table 1. The SRAM TLB is configured to use the same page sizes, capacity, and associativity as Coyote, making for a fair comparison. When either TLB may be used, FSRF selects the TLB with the best performance for the target workload size.

To obtain Coyote baselines, we port Coyote's TLB [39] to FSRF, and use it in lieu of FSRF's TLBs for address translation. We replicate

Table 1: Workloads used in our evaluation. Source indicates the primary source language, either (System)Verilog or Xilinx C/C++ HLS. TLB indicates whether FSRF used SRAM or DRAM for TLB storage, or which was used if the choice was dependent on dataset size. Small, large, and oversubscribed indicate the file sizes used in the respective experiments. Reuse reports how much data was accessed relative to the file size. Finally, reads reports the percentage of data read out of the total data accessed. Reuse and reads are data-dependent for PGRNK and reported as ranges for the datasets used.

Name	Description	Source	TLB	Dataset Sizes			Reuse	Reads
aes	AES-256 crypto	Verilog	SRAM	32 MiB	2 GiB	32 GiB	1x	50%
conv	Image convolution	Verilog	SRAM	32 MiB	2 GiB	32 GiB	1x	50%
dnn	AI inference	Verilog	SRAM	32.2 MiB	2.0 GiB	32.0 GiB	0.92x	96.7%
flow	Image optical flow	C HLS	SRAM	32 MiB	2 GiB	32 GiB	1x	100%
gups	HPCC RandomAccess	Verilog	S/D/D	32 MiB	2 GiB	2 GiB	4x	50%
hll	HyperLogLog	C HLS	SRAM	32 MiB	2 GiB	32 GiB	1x	100%
md5	MD5 hashing	Verilog	SRAM	32 MiB	2 GiB	32 GiB	1x	100%
nw	Needleman-Wunsch	Verilog	DRAM	32.2 MiB	2.0 GiB	32.0 GiB	5.0x	80.0%
pgrnk	Iterative PageRank	C HLS	S/S/D	54.2 MiB	3.1 GiB	35.5 GiB	1.18 - 1.86x	86.0 - 98.5%
rng	Random sequence	Verilog	S/D/D	32 MiB	2 GiB	32 GiB	1x	0-100%
sha	SHA-256 hashing	Verilog	SRAM	32 MiB	2 GiB	32 GiB	1x	100%
sha hls	SHA-256 hashing	C HLS	SRAM	32 MiB	2 GiB	32 GiB	1x	100%
tri	Triangle counting	C HLS	S/D/D	32 MiB	2 GiB	2 GiB	177.x	100%

many details of their design based on its open source code, including its latency and the default associativity and size of the TLB's (4×1024 4 KiB entries and 2 × 64 2 MiB entries). While Coyote allows for resizing its TLB, we opt not to as limited on-chip SRAM prevents expanding the 4 KiB TLB from having any benefit, and the 2 MiB TLB would need to be scaled up by 1-2 orders of magnitude for meaningful improvement. We also implement Coyote's striping policy by splitting 2 MiB regions across all 4 DIMMs. While not an end-to-end replica of Coyote's design, we believe these represent key aspects of their design and enable comparisons to similar FPGA virtualization systems that makes use of SRAM TLBs.

AmorphOS uses segment-based address translation to securely partition FPGA DRAM among resident applications. Since AmorphOS does not implement demand paging support, data must be loaded prior to use and memory cannot be oversubscribed. We replicate this approach by adding support for 4 GiB segments, which are large enough to cover our small and large datasets that fit in FPGA-local memory. We evaluate our AmorphOS baseline only on these datasets due to the lack of support for oversubscription. To simulate cold and warm data, we load the entire dataset on first access, replicating AmorphOS' requirement for preloading FPGA DRAM with data before use. Since the AmorphOS Memory Interface only allows fixed-size requests, we add hardware to convert multi-word AXI transactions into unique memory requests before forwarding them to the interconnect. Finally, since AmorphOS stripes accesses across DIMMs, we also implement this behavior and use the same striping granularity as our Coyote replica (2 MiB) for better comparison across the two systems.

Finally, our physical memory baseline elides virtual memory support for direct, unprotected access to FPGA-local DRAM, providing an upper bound on performance comparable to native implementation. Our measurements of the physical baseline are for hot data only, as demand paging and oversubscription require virtual memory to function.

6.3 Workload Results

Our results are shown in Table 2 and Figure 5, and reveal a number of trends and insights. For hot data, we find that FSRF consistently outperforms Coyote in streaming workloads (geomean +42%), despite using a similarly-configured SRAM TLB, and has almost double the performance of AmorphOS (geomean +95%). This is primarily an artifact of striping policy for Coyote, which can lead to FPGA DIMM collisions that FSRF's policy avoids. AmorphOS introduces additional inefficiencies with its memory interface, which cannot efficiently encode streaming accesses like FSRF's AXI4 interface. FSRF also matches physical memory performance in this case (geomean 98%) and achieves impressive levels of system throughput (geomean 51 GiB/s).

For large, non-streaming workloads, FSRF's DRAM TLB provides significant gains over Coyote (geomean +505%), moderate benefits versus AmorphOS (geomean +39%), and a large fraction of physical addressing performance (geomean 82%). These differences are more pronounced in workloads like GUPS, which has a large working set, low locality, and a small access size. On the other hand, PGRNK only randomly accesses a small segment of its data, giving it a relatively high SRAM TLB hit rate for a heavy random access workload.

For cold and warm data, FSRF performs similarly to Coyote and AmorphOS on small datasets (geomean +5%, +15%), due to high workload and system initiation overheads. However, large datasets enable significant performance differences over Coyote in non-streaming workloads (geomean +89%), due to FSRF's DRAM TLB again. We also find FSRF provides significant performance gains when memory is oversubscribed, especially in PGRNK (geomean +6, 114%) and non-streaming benchmarks in general (geomean +264%). For PGRNK, this is mostly due to input weights fitting in FPGA DRAM, enabling fast random access to frequently reused data while other graph data is streamed to and from FPGA memory. Other performance differences are largely due to FSRF's ability to dynamically adapt I/O batch size based on access patterns, as well as faster access to data present in FPGA DRAM.

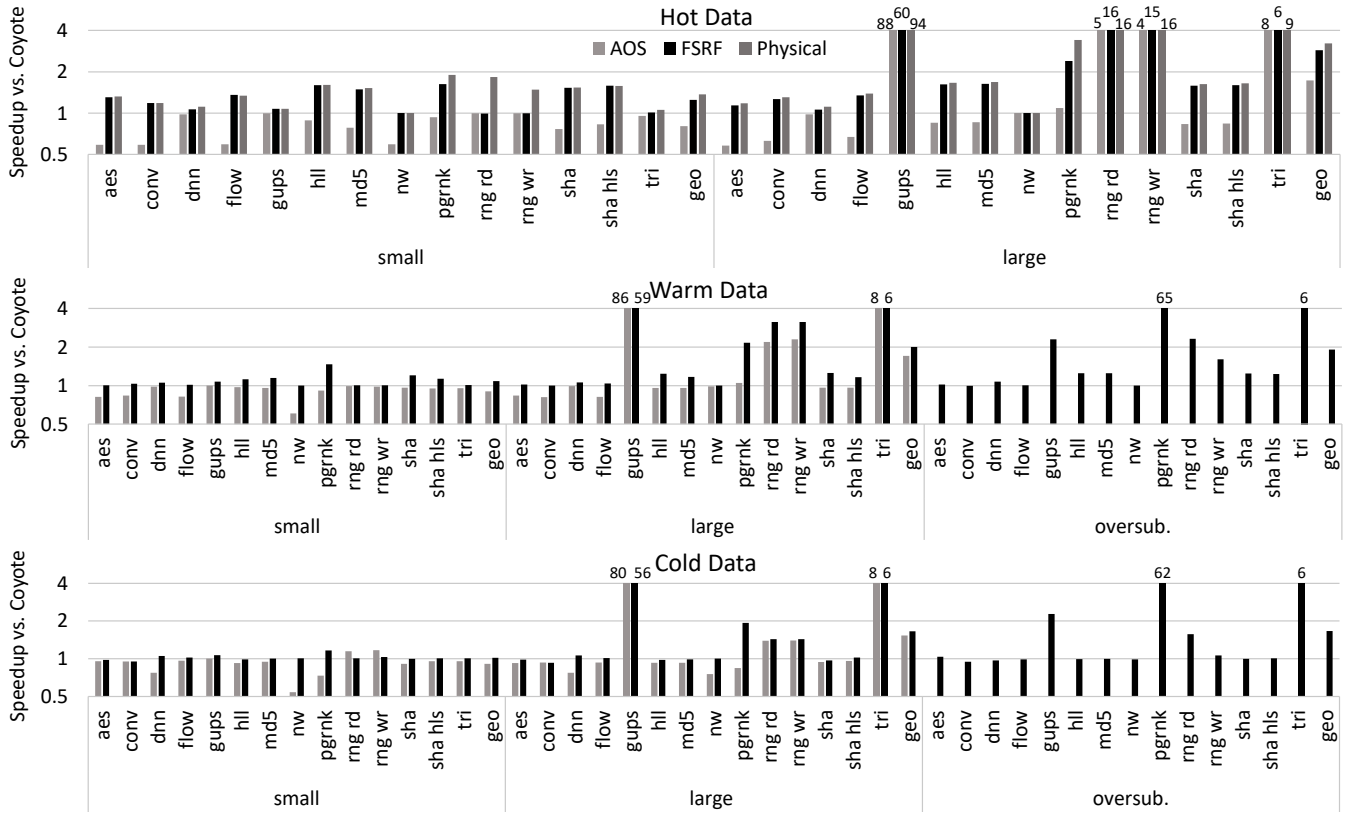

Figure 5: Average throughput of 4 parallel workloads, normalized to that of Coyote. The Coyote baseline uses 4 KiB or 2 MiB pages, whichever performs better; AmorphOS uses segments that cover the entire dataset; FSRF uses our system with either a DRAM or SRAM TLB; and physical accesses FPGA DRAM directly. Cold, warm, and hot denote whether file data starts on the host SSD, host DRAM, or FPGA DRAM. Small, large, and oversubscribed indicate the dataset size, with the latter exceeding an application's share of FPGA DRAM. Geo shows the geometric mean of all our benchmarks.

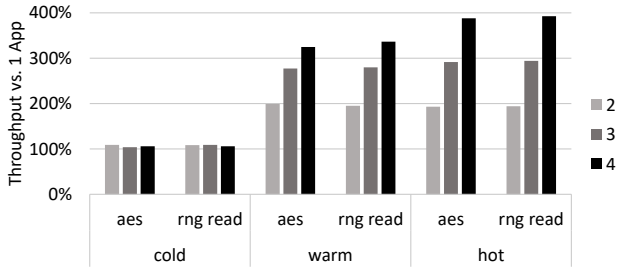

Figure 6: Performance scaling on large datasets with varying number of concurrent applications.

6.4 Performance Scaling and Fairness

Figure 6 shows how FSRF's throughput scales with the number of concurrent applications. We evaluate performance using AES and RNG with large datasets, representing both streaming and non-streaming workloads utilizing FSRF's SRAM and DRAM TLBs. We average data across 3 runs, normalize against the single-application case, and only report RNG read results due to them being comparable with RNG write ones.

We find that throughput scales almost linearly up to the point that FSRF becomes limited by available bandwidth. Accessing warm data limits performance gains beyond 3 concurrent applications, and cold data limits gains beyond just one active application. We also find that adding additional accelerators after throughput has been saturated is not detrimental to overall performance, with bandwidth being split between running applications.

To measure bandwidth-sharing fairness, we compute the standard deviation of individual application runtimes for each case, and normalize against their average. We find the deviations to be very low (geomean 1.1%), indicating fair sharing. While we do not find a trend in fairness with the number of applications (geomean 2: 1.2%, 3: 1.8%, and 4: 0.7%), we do see trends in application / TLB type (geomean AES: 0.7%, RNG read: 1.9%) and data "temperature" (geomean cold: 0.9%, warm: 2.7%, hot: 2.9%). In the former case, deviations are made worse by interleaving TLB accesses with random application accesses. And in the latter case, at least some of the unfairness, especially for hot data, is due to FSRF partitioning FPGA DIMMs among applications along with one of the DIMMs being roughly 6% slower than the others.

Table 2: Average end-to-end runtime (RT) and system throughput (TP) of 4 parallel workloads in seconds and MiB/s. Geom stream shows the geometric mean of our streaming workload results. Bolded values indicate whether FSRF, AmorphOS, or Coyote is faster, which can differ based on the metric used (see § 6.2). Values suffixed with k and m indicate units of 1000 and 1/1000, respectively.

		Cold			Warm			Hot	
Workload	System	Small RT/TP	Large RT/TP	Oversub. RT/TP	Small RT/TP	Large RT/TP	Oversub. RT/TP	Small RT/TP	Large RT/TP
dnn	**FSRF**	**.20/632**	**12./661**	222/573	**.18/684**	**11./686**	**189/672**	**.17/750**	**11./748**
	Cy 4K	.28/426	17./442	283/428	.21/551	14./550	224/539	.18/669	12./636
	Cy 2M	.20/602	12./621	**208/592**	.19/648	12./647	196/625	.17/705	11./703
	AOS	.26/465	16./479		.19/637	12./640		.17/690	11./689
	Phys							.16/785	10./784
gups	**FSRF**	**1.0/1.0k**	98./694	**2.6k/26.**	**.96/1.1k**	94./727	**2.6k/26.**	**.94/1.1k**	93./734
	Cy 4K	42./24.	5.3k/12.	5.9k/11.	42./25.	5.4k/12.	5.9k/11.	42./25.	5.4k/12.
	Cy 2M	1.0/980	8.9k/7.4	130k/.51	.98/1.0k	8.8k/7.4	133k/.49	.97/1.1k	8.8k/7.4
	AOS	1.1/981	**66./994**		.99/1.0k	**62./1.1k**		.97/1.1k	**61./1.1k**
	Phys							.95/1.1k	60./1.1k
nw	**FSRF**	**.19/11k**	**12./12k**	225/10k	**.18/12k**	11./12k	195/**12k**	**.17/13k**	**11./13k**
	Cy 4K	.25/9.1k	15./9.4k	279/8.1k	.23/9.5k	15./9.5k	272/8.3k	.18/12k	12./12k
	Cy 2M	.19/11k	12./12k	**219/10k**	.18/12k	**11./12k**	**194**/12k	.17/13k	11./13k
	AOS	.37/6.1k	16./8.9k		.30/7.3k	12./12k		.28/7.7k	11./13k
	Phys							.17/13k	11./13k
page rank	**FSRF**	**.16/1.9k**	**11./1.5k**	**139/2.0k**	**.10/2.9k**	**9.1/1.7k**	**132/2.1k**	**76u/3.7k**	**7.9/2.0k**
	Cy 4K	.29/1.0k	31./493	20k/14.	.26/1.2k	28./539	20k/14.	.19/1.5k	25./595
	Cy 2M	.18/1.6k	20./757	8.3k/33.	.15/2.0k	19./787	8.2k/33.	.13/2.3k	19./814
	AOS	.25/1.2k	24./636		.16/1.8k	18./824		.14/2.1k	17./886
	Phys							68u/4.3k	5.7/2.8k
rng read	**FSRF**	.10/1.4k	**5.2/1.6k**	**117/1.1k**	**27u/4.8k**	**1.0/8.1k**	**37./3.5k**	13u/9.8k	**.16/54k**
	Cy 4K	1.6/81.	20./409	185/722	1.4/95.	5.8/1.4k	88./1.5k	1.4/98.	2.8/3.0k
	Cy 2M	.10/1.4k	7.4/1.1k	283/464	27u/4.8k	3.2/2.6k	214/613	**13u/9.9k**	2.4/3.5k
	AOS	**91u/1.6k**	5.4/1.6k		27u/4.8k	1.6/5.7k		13u/9.9k	.47/18k
	Phys							7.2u/18k	.15/57k
rng write	**FSRF**	.11/1.3k	**5.2/1.6k**	**210/649**	**26u/4.9k**	**1.0/8.1k**	**78./1.8k**	13u/9.9k	**.17/51k**
	Cy 4K	1.7/79.	20./415	221/612	1.4/97.	5.6/1.5k	126/1.1k	1.3/100	2.8/3.0k
	Cy 2M	.11/1.2k	7.5/1.1k	1.6k/91.	26u/4.9k	3.2/2.6k	1.7k/82.	**13u/9.9k**	2.5/3.4k
	AOS	**.10/1.5k**	5.5/1.6k		27u/4.8k	1.4/5.9k		13u/9.8k	.54/15k
	Phys							8.9u/15k	.16/54k
tri	**FSRF**	3.7/**1.6k**	371/1.0k	**472/802**	3.7/**1.6k**	366/1.0k	**465/814**	3.7/**1.6k**	366/1.0k
	Cy 4K	13./455	2.7k/135	2.7k/135	12./464	2.7k/135	2.7k/136	12./465	2.7k/135
	Cy 2M	**3.6**/1.6k	2.0k/186	3.2k/115	**3.6**/1.6k	2.0k/185	3.2k/115	**3.6**/1.6k	2.0k/186
	AOS	3.8/1.5k	**244/1.5k**		3.7/1.5k	**239/1.5k**		3.7/1.5k	**239/1.5k**
	Phys							3.5/1.7k	226/1.7k
geom stream	**FSRF**	**86u/1.7k**	**5.4**/1.6k	**96./1.4k**	**16u/8.3k**	**.91/9.0k**	**16./8.3k**	**2.8u/49k**	**.15/55k**
	Cy 4K	.11/1.3k	6.0/1.4k	108/1.2k	90u/1.5k	5.5/1.5k	93./1.4k	32u/4.2k	2.8/3.0k
	Cy 2M	86u/1.7k	5.4/**1.6k**	97./1.4k	17u/7.6k	1.0/8.0k	18./7.3k	4.0u/34k	.21/38k
	AOS	90u/1.6k	5.6/1.5k		19u/6.8k	1.1/7.2k		5.5u/24k	.29/29k
	Phys							2.8u/49k	.15/57k

6.5 Mixed Workload Fairness

We evaluate fairness for a heterogeneous configuration of workloads, representing an accelerator being shared concurrently by different users. We use a diverse selection of memory-intensive applications: AES, PGRNK, SHA HLS, and RNG. These are evaluated with hot data from the large dataset for each. Since runtimes between applications vary significantly, we measure throughput by running PGRNK 25 times and then waiting for the other workloads to finish, resulting in the system being fully saturated for at least 99.4% of the end-to-end runtime. To understand how performance is affected by a mixture of concurrent workloads sharing the FPGA, we report their throughput relative to it when running individually under the same system. Results are shown in Figure 7.

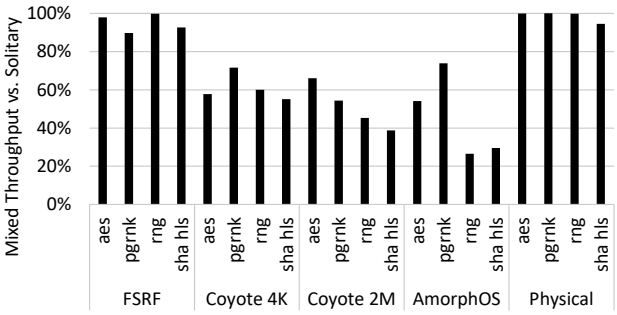

Figure 7: Relative per-workload throughput under contended, heterogeneous operation vs. solitary execution.

Table 3: Resource utilization of FSRF's main components, vendor IPs, and workloads.

Entity	LUT	FF	BRAM	URAM
Xbar	17,455	968	0	0
D-TLB	3,310	1,150	0	0
S-TLB	1,402	247	14	0
DMA	4,629	1,263	8	8
XDMA	63,342	62,527	123	0
DDR4	26,216	25,691	25	0
aes	42,687	36,330	289	0
conv	9,453	32,715	0	16
dnn	8,781	4,587	6	0
flow	34,613	61,124	86	0
gups	966	651	0	0
hll	23,849	44,189	168	0
md5	10,282	12,757	0	0
nw	133,798	78,294	15	0
pgrnk	20,723	29,958	31	0
rng	1,612	1,303	0	0
sha	20,683	42,271	0	0
sha hls	28,739	57,439	8	0
tri	7,873	13,357	160	0
FPGA	1,182,240	2,364,480	2,160	960

We find that both FSRF and the physical baseline provide good performance isolation relative to Coyote and AmorphOS, with applications enjoying similar performance regardless of what other applications are running concurrently, due to DIMMs being partitioned among applications. Coyote and AmorphOS' striping policies result in unfair sharing that causes inconsistent performance when the FPGA is loaded with different applications. With Coyote (2 MiB pages), for instance, AES achieves 66% of solitary performance, while SHA HLS only achieves 39% of its throughput baseline. With AmorphOS, PGRNK and RNG see very different performance under contention relative to solitary operation (74% vs. 27%, respectively). These results demonstrate the importance of flexible striping policies when sharing access to memory resources, as more naive approaches can result in unfairness, especially in heterogeneous configurations.

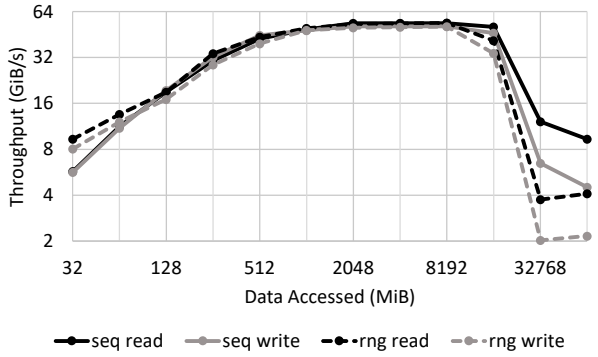

Figure 8: FSRF DRAM TLB performance for repeated accesses of various sizes to cached data.

6.6 Resource Usage

We report the average resource usage of various FPGA components in Table 3. Resources include look-up tables, flip-flops, block RAMs, and UltraRAMs, with the final row indicating the total number present in each F1 FPGA.

For FSRF, we measure our most significant components: the custom crossbar for the interconnect, our DRAM and SRAM TLB modules, and our custom DMA implementation. We also record results for vendor IPs, including the vendor DMA module present in the F1 shell and the DDR4 memory controllers we instantiate in user logic. We include data for our various workloads as well.

We find that overall, FSRF introduces very reasonable overheads for the functionality it provides. Our crossbar, 4 DRAM and SRAM TLBs, and our DMA module only consume 3.5% of FPGA LUTs available, 0.3% of FFs, 3% of BRAMs, and 1% of URAMS. The 8 URAMs aside, these use far fewer resources than a single vendor DMA module, the cost paid for the 3 user-instantiated DDR4 controllers on F1, or even just a single instance of AES.

6.7 AXI Access Size Microbenchmark

Figure 8 shows how access size, ranging from 64 B to 4 KiB, affects FSRF's performance when performing a constant number of accesses to cached data with a DRAM TLB. We find FSRF works effectively in most cases, but is limited by memory performance for smaller accesses, and memory capacity for larger footprints.

6.8 Transfer Size Microbenchmark

Figure 9 shows the transfer performance for various I/O sizes ranging from 4 KiB to 2 MiB for a random access pattern from a single application. Data is shown for how quickly FSRF can fault in warm and cold data at each size, along with how quickly the host can read from the NVMe drive and DMA data to the FPGA. We find that FSRF generally performs within 2× of whichever component limits throughput, with file access and DMA dominating run time.

7 RELATED WORK

FPGA OS and Virtualization. Research on FPGA OSes includes sharing FPGA fabric [22, 27, 38, 63, 64, 113], multiplexing [28, 40,

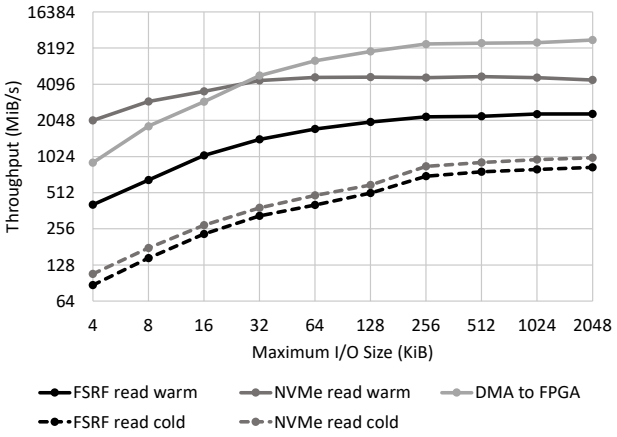

Figure 9: FSRF performance for various I/O batch sizes.

102, 111], context switch [70, 94], memory virtualization [1, 30, 75, 117], relocation [53], preemption [72], and heterogeneous execution [20, 43, 102, 111]. FPGA OS abstractions [27, 36, 75, 78], FPGA hypervisor support [35, 80, 87, 89], and FPGA virtualization based on overlays [19, 56, 61, 62, 65, 115] are rich research areas. ViTAL [118–120] supports a single-FPGA abstraction over multiple FPGAs, but does not consider I/O through the OS. AmorphOS [60]and FOS [109] provide OS-level sharing of FPGAs among mutually distrustful processes: I/O support is mediated with fixed techniques that encapsulate existing interfaces. Coyote [66] supports communication and virtual memory management for FPGAs. Like FSRF, Coyote customizes virtual memory implementation to applications, but only by configuring TLB geometry, which can limit performance (§6). Cascade [96] and Synergy [68] virtualize FPGAs at the language interface: Synergy's techniques could be used to implement higher-level `libc`-like interfaces in device-side code atop FSRF's lower-level I/O interfaces.

Accelerators and OS-Managed Resources. Prior work has explored exposing file systems [100] and the syscall interface [75, 100] to FPGAs as well as memory virtualization for FPGAs [1, 30, 75, 117]. GPUfs [98] exposed the file system interface and page cache to GPUs, and GAIA [21] used GPU memory to extend the page cache. Vesely et al. further expanded on those techniques [110] to explore the implications of supporting the entire system call interface to GPUs. FSRF improves upon these techniques by using an `mmap`-like interface to enable file system accesses to complete on-device via virtual memory, and by using reconfigurability and adaptable policies to tailor its implementation to application needs.

Accelerators and Virtual Memory. Address translation overheads are well-understood for CPUs [2, 3, 14, 15, 17, 18, 41, 42, 54, 58, 76, 83, 85, 86, 95, 101, 105] and GPUs [31, 88, 90] MASK [12] is a TLB-aware GPU memory hierarchy design extended by Mosaic [11] to provide application-transparent multiple page size support in GPUs, further extended by ETC [73] to optimize oversubscription. Modern GPUs automate GPU memory management: pages are moved to/from GPU memory on-demand, and kernel execution overlaps data transfer, based on techniques proposed by Zheng et al. [122]. GPUswap [59] enables GPU memory oversubscription by relocating GPU application data to CPU memory, keeping data accessible from the GPU. FSRF uses similar techniques to move for over-subscription, but our focus is on FPGAs.

8 CONCLUSION

FSRF demonstrates that FPGA I/O can be abstracted in a familiar, performant, and portable way through `mmap`-style file access and on-device virtual memory address translation. We address limitations of current approaches by using reconfiguration and adaptable software policies to dynamically customize FSRF's implementation to application needs, even during runtime. We evaluate our design with a wide variety of workloads and find it significantly improves performance over state-of-the-art virtual memory techniques and has minimal overheads relative to a physical addressing upper-bound.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback, insightful comments, and guidance that helped us improve this paper. This research was supported by NSF grants CNS-1846169, CNS-2006943, the Texas Systems Research Consortium, and the National Nuclear Security Administration Award Number DE-NA0003969

A ARTIFACT APPENDIX

A.1 Abstract

This artifact appendix documents the requirements and instructions for setting up FSRF and reproducing the results in this paper. FSRF runs on Amazon AWS F1 cloud FPGA infrastructure. We provide both code and scripts to setup and use FSRF, to download the openly-available data used by PageRank, and to generate the inputs used in our evaluation. We provide the source code for FSRF and scripts to aid in system setup and evaluation at: https://github.com/utcs-scea/amorphos-fsrf.

A.2 Artifact Checklist

Checklist details: https://ctuning.org/ae/checklist.html

- **Algorithm**: FPGA file-backed virtual-memory runtime.
- **Program**: Our repo includes assorted hardware workloads with corresponding software drivers.
- **Compilation**: The AWS FPGA Dev AMI includes a C++ compiler for host software and access to Xilinx Vivado for FPGA hardware compilation.
- **Binary**: Our code is compiled from source. FPGA software binaries are provided by Amazon with their FPGA Dev AMI.
- **Data set**: PageRank uses datasets from the SuiteSparse Matrix Collection and TRI uses generated graphs. We provide scripts to download and generate binaries for these workloads, which use a total of ~85 GiB of storage. Our other workloads use datasets generated directly on the instance's ephemeral SSD.
- **Run-time environment**: AWS FPGA Dev AMI 1.10.
- **Hardware**: AWS EC2 F1 instances.
- **Run-time state**: FSRF utilizes the host page cache. Our provided scripts clear this cache when appropriate.
- **Execution**: Most workloads can be compiled in ~4 hours, with a few taking up to a couple hours longer. FPGA execution often takes seconds to minutes, but can take hours when oversubscribed or simulating a sub-optimal system.

- **Metrics**: Application software reports data accessed, runtimes, and throughput for each run. The FSRF daemon can also print runtime metadata. Reports on design resource utilization can be obtained from Vivado for FPGA builds.
- **Output**: Binaries provide data through the console. Our scripts log data to a file and print condensed results.
- **Experiments**: We provide a README and scripts to cover much of the setup and evaluation process.
- **How much disk space required?**: The AWS FPGA Dev AMI uses ~50 GiB. The graph files use 85 GiB. FPGA builds can use ~2 GiB each. Our code and binaries use ~10 MiB.
- **How much time is needed to prepare workflow?**: Setting up the F1 environment takes about 1–2 hours. Sequentially compiling all bitstreams should take around 3 – 4 days.
- **How much time is needed to complete experiments?**: Running all the workloads in our main experiment once takes about 6 days. Our other experiments could be run in an hour.
- **Publicly available?**: Yes.
- **Code licenses?**: BSD-2.
- **Workflow framework used?**: Our code provides interfaces to aid in scripting our experiments.

A.3 Description

A.3.1 How to Access. Our artifact can be obtained by cloning the repository in the abstract. The artifact does not require much space on its own, but running the first-time setup script will require about 85 GiB of data.

A.3.2 Hardware Dependencies. FSRF runs on AWS EC2 F1 instances, including the f1.2xlarge. We recommend using f1.4xlarge instance for replicating our oversubscription results, which utilize the additional RAM for caching the large dataset.

A.3.3 Software Dependencies. The proprietary AWS FPGA Developer AMI provides all the software tools needed to compile, manage, and interface with FPGA applications. This toolkit is currently provided to AWS users at no additional cost.

A.3.4 Datasets. We use open datasets from the SuiteSparse Matrix Collection [34]. These include the Matrix Market versions of webbase-1M [116], mawi_201512020030 [8], and GAP-kron [16]. Users can download the data via a provided script or obtain it from the SuiteSparse website directly.

A.4 Installation

The README in our repository covers the installation process for users who are already set up to use AWS F1. We also include a first-time setup script, which automates much of the process on new instances.

A.5 Experiment Workflow

The README documents our main experiment workflow, including how to set up the input files, start the daemon, and run scripts that automate our major experiments. It also includes information on our application software for those who wish to run additional experiments.

A.6 Evaluation and Expected Results

We provide experiment scripts, which record their results to log files for evaluation. Application binaries support even more run customization arguments and print their data to the console to aid in running smaller experiments. Applications report their runtime, throughput, and other metrics, which can be compared with the data in this paper to verify results.

A.7 Experiment Customization

Most of our benchmarks can be modified to operate on different binary inputs by changing the filename and dataset size in their software drivers. The binaries also support a wide variety of input arguments for further customization. Our artifact additionally includes HDL for those wishing to utilize FSRF's functionality with their own hardware accelerators.

REFERENCES

[1] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. 2011. Leap Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. ACM, New York, NY, USA, 25–28. https://doi.org/10.1145/1950413.1950421

[2] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2012. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. In *ISCA*.

[3] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2015. Fast Two-Level Address Translation for Virtualized Systems. In *IEEE TC*.

[4] Alibaba Corp. 2021. Alibaba Cloud Instances Overview. https://www.alibabacloud.com/help/doc-detail/108504.htm.

[5] Altera. [n. d.]. Integrating 100-GbE Switching Solutions on 28nm FPGAs. https://www.altera.com/en_US/pdfs/literature/wp/wp-01127-stxv-100gbe-switching.pdf. (Accessed May 2018).

[6] Altera. 2018. Cyclone V SoC Development Board Reference Manual. https://www.altera.com/en_US/pdfs/literature/manual/rm_cv_soc_dev_board.pdf. (Accessed May 2018).

[7] Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, Ron Sass, and David Andrews. 2006. Enabling a Uniform Programming Model across the Software/Hardware Boundary. FCCM 06.

[8] MAWI Working Group Traffic Archive. [n. d.]. MAWI/mawi_201512020030. https://sparse.tamu.edu/MAWI/mawi_201512020030. (Accessed on 06/27/2022).

[9] ARM. [n. d.]. Cortex-A15 MPCore Processor Technical Reference Manual. https://developer.arm.com/documentation/ddi0438/i/memory-management-unit/tlb-organization/l2-tlb?lang=en.

[10] ARM. 2013. AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite. https://developer.arm.com/documentation/ihi0022/e/. (Accessed on 4/30/2021).

[11] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *MICRO*.

[12] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. 2018. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, New York, NY, USA, 503–518. https://doi.org/10.1145/3173162.3173169

[13] Baidu Corp. 2021. Baudi Cloud Instances Overview. https://cloud.baidu.com/product/fpga.html.

[14] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don'T Walk (the Page Table). In *ISCA*.

[15] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *ISCA*.

[16] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[17] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared Last-level TLBs for Chip Multiprocessors. In *HPCA*.

[18] Abhishek Bhattacharjee and Margaret Martonosi. 2009. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In *PACT*.

[19] Alexander Brant and Guy GF Lemieux. 2012. ZUMA: An open FPGA overlay architecture. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 93–96.

[20] Gordon J. Brebner. 1996. A Virtual Hardware Operating System for the Xilinx XC6200. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers (FPL '96)*. Springer-Verlag, London, UK, UK, 327–336. http://dl.acm.org/citation.cfm?id=647923.741195

[21] Tanya Brokhman, Pavel Lifshits, and Mark Silberstein. 2019. GAIA: An OS Page Cache for Heterogeneous Systems. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 661–674.

[22] 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 *Proceedings of the 2014 IEEE 22Nd International Symposium*

on Field-Programmable Custom Computing Machines (FCCM '14). IEEE Computer Society, Washington, DC, USA, 109–116.

[23] Stuart Byma, Naif Tarafdar, Talia Xu, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2015. Expanding OpenFlow Capabilities with Virtualized Reconfigurable Hardware. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '15)*. ACM, New York, NY, USA, 94–97. https://doi.org/10.1145/2684746.2689086

[24] Jared Casper and Kunle Olukotun. 2014. Hardware Acceleration of Database Operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays* (Monterey, California, USA) *(FPGA '14)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/2554688.2554787

[25] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A Cloud-Scale Acceleration Architecture, In Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture. https://www.microsoft.com/en-us/research/publication/configurable-cloud-acceleration/

[26] Zhilei Chai, Jin Yu, Zhibin Wang, Jie Zhang, and Haojie Zhou. 2015. An Embedded FPGA Operating System Optimized for Vision Computing (Abstract Only). In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '15)*. ACM, New York, NY, USA, 271–271. https://doi.org/10.1145/2684746.2689127

[27] 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* (Cagliari, Italy) *(CF '14)*. ACM, New York, NY, USA, Article 3, 10 pages. https://doi.org/10.1145/2597917.2597929

[28] Liang Chen, Thomas Marconi, and Tulika Mitra. 2012. Online Scheduling for Multi-core Shared Reconfigurable Fabric. In *Proceedings of the Conference on Design, Automation and Test in Europe* (Dresden, Germany) *(DATE '12)*. EDA Consortium, San Jose, CA, USA, 582–585. http://dl.acm.org/citation.cfm?id=2492708.2492853

[29] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. IEEE. https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/

[30] 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* (Monterey, CA, USA) *(FPGA '11)*. ACM, New York, NY, USA, 97–106. https://doi.org/10.1145/1950413.1950435

[31] Jason Cong, Zhenman Fang, Yuchen Hao, and Glenn Reinmana. 2017. Supporting Address Translation for Accelerator-Centric Architectures. In *HPCA*.

[32] Louise H Crockett, Ross A Elliot, Martin A Enderwitz, and Robert W Stewart. 2014. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media.

[33] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '16)*. ACM, New York, NY, USA, 105–110. https://doi.org/10.1145/2847263.2847339

[34] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[35] André DeHon, Yury Markovsky, Eylon Caspi, Michael Chu, Randy Huang, Stylianos Perissakis, Laura Pozzi, Joseph Yeh, and John Wawrzynek. 2006. Stream computations organized for reconfigurable execution. *Microprocessors and Microsystems* 30, 6 (2006), 334–354. https://doi.org/10.1016/j.micpro.2006.02.009

[36] Alexander Domahidi, Eric Chu, and Stephen Boyd. 2013. ECOS: An SOCP solver for embedded systems. In *Control Conference (ECC), 2013 European*. IEEE, 3071–3076.

[37] Amazon EC2. 2017. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/

[38] Suhaib A. Fahmy, Kizheppatt Vipin, and Shanker Shreejith. 2015. Virtualized FPGA Accelerators for Efficient Cloud Computing. In *Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom) (CLOUDCOM '15)*. IEEE Computer Society, Washington, DC, USA, 430–435.

[39] ETH Zurich FPGA Systems Group. [n. d.]. fpgasystems/Coyote. https://github.com/fpgasystems/Coyote.

[40] W. Fu and K. Compton. 2008. Scheduling Intervals for Reconfigurable Computing. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*. 87–96. https://doi.org/10.1109/FCCM.2008.48

[41] Jayneel Gandhi, , Mark D. Hill, and Michael M. Swift. 2016. Exceeding the Best of Nested and Shadow Paging. In *ISCA*.

[42] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. Efficient Memory Virtualization. In *MICRO*.

[43] Ivan Gonzalez, Sergio Lopez-Buedo, Gustavo Sutter, Diego Sanchez-Roman, Francisco J. Gomez-Arribas, and Javier Aracil. 2012. Virtualization of Reconfigurable Coprocessors in HPRC Systems with Multicore Architecture. *J. Syst. Archit.* 58, 6-7 (June 2012), 247–256. https://doi.org/10.1016/j.sysarc.2012.03.002

[44] B. K. Hamilton, M. Inggs, and H. K. H. So. 2014. Scheduling Mixed-Architecture Processes in Tightly Coupled FPGA-CPU Reconfigurable Computers. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. 240–240. https://doi.org/10.1109/FCCM.2014.75

[45] John L. Hennessy and David A. Patterson. 2018. A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development. http://iscaconf.org/isca2018/turing_lecture.html.

[46] Homer Hsing. [n. d.]. Tiny AES. https://opencores.org/projects/tiny_aes. (Accessed on 09/20/2021).

[47] Chun-Hsian Huang and Pao-Ann Hsiung. 2009. Hardware Resource Virtualization for Dynamically Partially Reconfigurable Systems. *IEEE Embed. Syst. Lett.* 1, 1 (May 2009), 19–23. https://doi.org/10.1109/LES.2009.2028039

[48] Huawei Corp. 2021. FPGA Accelerated Cloud Server. https://www.huaweicloud.com/en-us/product/fcs.html.

[49] Apple Inc. 2013. About the Virtual Memory System. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/Articles/AboutMemory.html.

[50] Aws Ismail and Lesley Shannon. 2011. FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '11)*. IEEE Computer Society, Washington, DC, USA, 170–177. https://doi.org/10.1109/FCCM.2011.48

[51] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. 2016. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA) *(NSDI'16)*. USENIX Association, Berkeley, CA, USA, 425–438. http://dl.acm.org/citation.cfm?id=2930611.2930639

[52] Alexander Kaganov, Asif Lakhany, and Paul Chow. 2011. FPGA Acceleration of MultiFactor CDO Pricing. *ACM Trans. Reconfigurable Technol. Syst.* 4, 2, Article 20 (May 2011), 17 pages. https://doi.org/10.1145/1968502.1968511

[53] H. Kalte and M. Porrmann. 2005. Context saving and restoring for multitasking in reconfigurable systems. In *Field Programmable Logic and Applications, 2005. International Conference on*. 223–228. https://doi.org/10.1109/FPL.2005.1515726

[54] Gokul B. Kandiraju and Anand Sivasubramaniam. 2002. Going the Distance for TLB Prefetching: An Application-driven Study. In *ISCA*.

[55] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. ACM, New York, NY, USA, 295–308. https://doi.org/10.1145/2168836.2168866

[56] Nachiket Kapre and Jan Gray. 2015. Hoplite: Building austere overlay NoCs for FPGAs. In *FPL*. IEEE, 1–8.

[57] Kaan Kara and Gustavo Alonso. 2016. Fast and robust hashing for database operators. In *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016*. 1–4. https://doi.org/10.1109/FPL.2016.7577353

[58] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. ACM, New York, NY, USA, 66–78. https://doi.org/10.1145/2749469.2749471

[59] Jens Kehne, Jonathan Metter, and Frank Bellosa. 2015. GPUswap: Enabling Oversubscription of GPU Memory through Transparent Swapping. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Istanbul, Turkey) *(VEE '15)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/2731186.2731192

[60] 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)*. 107–127.

[61] Robert Kirchgessner, Alan D. George, and Greg Stitt. 2015. Low-Overhead FPGA Middleware for Application Portability and Productivity. *ACM Trans. Reconfigurable Technol. Syst.* 8, 4, Article 21 (Sept. 2015), 22 pages. https://doi.org/10.1145/2746404

[62] Robert Kirchgessner, Greg Stitt, Alan George, and Herman Lam. 2012. VirtualRC: A Virtual FPGA Platform for Applications and Tools Portability. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '12)*. ACM, New York, NY, USA, 205–208. https://doi.org/10.1145/2145694.2145728

[63] O. Knodel, P. Lehmann, and R. G. Spallek. 2016. RC3E: Reconfigurable Accelerators in Data Centres and Their Provision by Adapted Service Models. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. 19–26. https://doi.org/10.1109/CLOUD.2016.0013

[64] Oliver Knodel and Rainer G. Spallek. 2015. RC3E: Provision and Management of Reconfigurable Hardware Accelerators in a Cloud Environment. *CoRR* abs/1508.06843 (2015). arXiv:1508.06843

[65] Dirk Koch, Christian Beckhoff, and Guy G. F. Lemieux. 2013. An efficient FPGA overlay for portable custom instruction set extensions. In *FPL*. IEEE, 1–8.

[66] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 991–1010. https://www.usenix.org/conference/osdi20/presentation/roscoe

[67] A. Kulkarni, M. Chiosa, T. B. Preußer, K. Kara, D. Sidler, and G. Alonso. 2020. HyperLogLog Sketch Acceleration on FPGA. In *FPL*.

[68] Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J. Rossbach, and Eric Schkufza. 2021. Compiler-Driven FPGA Virtualization with SYNERGY. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 818–831. https://doi.org/10.1145/3445814.3446755

[69] Christian Leber, Benjamin Geib, and Heiner Litz. 2011. High Frequency Trading Acceleration Using FPGAs. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications (FPL '11)*. IEEE Computer Society, Washington, DC, USA, 317–322. https://doi.org/10.1109/FPL.2011.64

[70] Trong-Yen Lee, Che-Cheng Hu, Li-Wen Lai, and Chia-Chun Tsai. 2010. Hardware Context-Switch Methodology for Dynamically Partially Reconfigurable Systems. *J. Inf. Sci. Eng.* 26 (2010), 1289–1305.

[71] John Leitch. [n. d.]. MD5 Pipelined. https://opencores.org/projects/md5_pipelined. (Accessed on 09/20/2021).

[72] L. Levinson, R. Manner, M. Sessler, and H. Simmler. 2000. Preemptive multitasking on FPGAs. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*. 301–302. https://doi.org/10.1109/FPGA.2000.903426

[73] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A Framework for Memory Oversubscription Management in Graphics Processing Units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 49–63. https://doi.org/10.1145/3297858.3304044

[74] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. 2015. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. ACM, New York, NY, USA, 476–488. https://doi.org/10.1145/2749469.2750416

[75] Enno Lübbers and Marco Platzner. 2009. ReconOS: Multithreaded Programming for Reconfigurable Computers. *ACM Trans. Embed. Comput. Syst.* 9, 1, Article 8 (Oct. 2009), 33 pages. https://doi.org/10.1145/1596532.1596540

[76] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. 2013. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. *ACM TACO* (2013).

[77] Roman Lysecky, Kris Miller, Frank Vahid, and Kees Vissers. 2005. Firm-core Virtual FPGA for Just-in-Time FPGA Compilation (Abstract Only). In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays* (Monterey, California, USA) *(FPGA '05)*. ACM, New York, NY, USA, 271–271. https://doi.org/10.1145/1046192.1046247

[78] 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 827–844. https://doi.org/10.1145/3373376.3378482

[79] Microsoft. 2017. Microsoft Azure Goes Back To Rack Servers With Project Olympus. https://www.nextplatform.com/2016/11/01/microsoft-azure-goes-back-rack-servers-project-olympus/. (Accessed July 2018).

[80] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. 2006. Tartan: Evaluating Spatial Computation for Whole Program Execution. *SIGOPS Oper. Syst. Rev.* 40, 5 (Oct. 2006), 163–174. https://doi.org/10.1145/1168917.1168878

[81] NEC. [n. d.]. neoface | NEC Today. http://nectoday.com/tag/neoface/. (Accessed April 2019).

[82] Tayo Oguntebi and Kunle Olukotun. 2016. GraphOps: A Dataflow Library for Graph Analytics Acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '16)*. ACM, New York, NY, USA, 111–117. https://doi.org/10.1145/2847263.2847337

[83] M.-M. Papadopoulou, Xin Tong, A. Seznec, and A. Moshovos. 2015. Prediction-based Superpage-friendly TLB Designs. In *HPCA*.

[84] Wesley Peck, Erik K. Anderson, Jason Agron, Jim Stevens, Fabrice Baijot, and David L. Andrews. 2006. Hthreads: A Computational Model for Reconfigurable Devices.. In *FPL*. IEEE, 1–4.

[85] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. Increasing TLB Reach by Exploiting Clustering in Page Translations. In *HPCA*.

[86] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *MICRO*.

[87] K. Dang Pham, A. K. Jain, J. Cui, S. A. Fahmy, and D. L. Maskell. 2013. Microkernel hypervisor for a hybrid ARM-FPGA platform. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*. 219–226. https://doi.org/10.1109/ASAP.2013.6567578

[88] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *ASPLOS*.

[89] Christian Plessl and Marco Platzner. 2005. Zippy-A coarse-grained reconfigurable array with support for hardware virtualization. In *Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE International Conference on*. IEEE, 213–218.

[90] Jason Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 568–578. https://doi.org/10.1109/HPCA.2014.6835965

[91] progranism, teknohog, OrphanedGland, udif, TheSeven, makomk, and newMeat1. [n. d.]. Open Source FPGA Bitcoin Miner. https://github.com/progranism/Open-Source-FPGA-Bitcoin-Miner. (Accessed on 10/24/2017).

[92] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *41st Annual International Symposium on Computer Architecture (ISCA)*. http://research.microsoft.com/apps/pubs/default.aspx?id=212001

[93] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2016. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. *Commun. ACM* 59, 11 (Oct. 2016), 114–122. https://doi.org/10.1145/2996868

[94] Kyle Rupnow, Wenyin Fu, and Katherine Compton. 2009. Block, Drop or Roll(back): Alternative Preemption Methods for RH Multi-Tasking. In *FCCM 2009, 17th IEEE Symposium on Field Programmable Custom Computing Machines, Napa, California, USA, 5-7 April 2009, Proceedings*. 63–70. https://doi.org/10.1109/FCCM.2009.30

[95] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. 2000. Recency-based TLB Preloading. In *ISCA*.

[96] 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, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. 271–286.

[97] 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*.

[98] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[99] Hayden Kwok-Hay So and Robert Brodersen. 2008. A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers Using BORPH. *ACM Trans. Embed. Comput. Syst.* 7, 2, Article 14 (Jan. 2008), 28 pages. https://doi.org/10.1145/1331331.1331338

[100] Hayden Kwok-Hay So and Robert W. Brodersen. 2007. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. Ph. D. Dissertation. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-92.html

[101] Shekhar Srikantaiah and Mahmut Kandemir. 2010. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. In *MICRO*.

[102] C. Steiger, H. Walder, and M. Platzner. 2004. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Trans. Comput.* 53, 11 (Nov 2004), 1393–1407. https://doi.org/10.1109/TC.2004.99

[103] G. Stitt and J. Coole. 2011. Intermediate Fabrics: Virtual Architectures for Near-Instant FPGA Compilation. *IEEE Embedded Systems Letters* 3, 3 (Sept 2011), 81–84. https://doi.org/10.1109/LES.2011.2167713

[104] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '16)*. ACM, New York, NY, USA, 16–25. https://doi.org/10.1145/2847263.2847276

[105] M. Talluri and M. D. Hill. 1994. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *ASPLOS*.

[106] Mellanox Technologies. 2018. Innova-2 Flex Programmable Network Adapter. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf. (Accessed May 2018).

[107] Tencent Corp. 2021. FPGA Cloud Server. https://cloud.tencent.com/product/fpga.

[108] A. Tsutsui, T. Miyazaki, K. Yamada, and N. Ohta. 1995. Special Purpose FPGA for High-speed Digital Telecommunication Systems. In *Proceedings of the 1995 International Conference on Computer Design: VLSI in Computers and Processors (ICCD '95)*. IEEE Computer Society, Washington, DC, USA, 486–491. http://dl.acm.org/citation.cfm?id=645463.653355

[109] Anuj Vaishnav, Khoa Dang Pham, Joseph Powell, and Dirk Koch. 2020. FOS: A Modular FPGA Operating System for Dynamic Workloads. *ACM Trans. Reconfigurable Technol. Syst.* 13, 4, Article 20 (sep 2020), 28 pages. https://doi.org/10.1145/3405794

[110] Ján Veselý, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H. Loh, Mark Oskin, and Steven K. Reinhardt. 2018. Generic System Calls for GPUs. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 843–856. https://doi.org/10.1109/ISCA.2018.00075

[111] G. Wassi, Mohamed El Amine Benkhelifa, G. Lawday, F. Verdier, and S. Garcia. 2014. Multi-shape tasks scheduling for online multitasking on FPGAs. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*. 1–7. https://doi.org/10.1109/ReCoSoC.2014.6861366

[112] Matthew A. Watkins and David H. Albonesi. 2010. ReMAP: A Reconfigurable Heterogeneous Multicore Architecture. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 497–508. https://doi.org/10.1109/MICRO.2010.15

[113] Jagath Weerasinghe, François Abel, Christoph Hagleitner, and Andreas Herkersdorf. 2015. Enabling FPGAs in Hyperscale Data Centers. In *2015 IEEE 12th Intl Conf on 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), Beijing, China, August 10-14, 2015*. 1078–1086.

[114] Stephan Werner, Oliver Oey, Diana Göhringer, Michael Hübner, and Jürgen Becker. 2012. Virtualized On-chip Distributed Computing for Heterogeneous Reconfigurable Multi-core Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe* (Dresden, Germany) *(DATE '12)*. EDA Consortium, San Jose, CA, USA, 280–283. http://dl.acm.org/citation.cfm?id=2492708.2492778

[115] Tobias Wiersema, Ame Bockhorn, and Marco Platzner. 2014. Embedding FPGA overlays into configurable Systems-on-Chip: ReconOS meets ZUMA. In *ReConFig*. IEEE, 1–6.

[116] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2009. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Comput.* 35, 3 (2009), 178–194. https://doi.org/10.1016/j.parco.2008.12.006 Revolutionary Technologies for Acceleration of Emerging Petascale Applications.

[117] Felix Winterstein, Kermin Fleming, Hsin-Jung Yang, Samuel Bayliss, and George Constantinides. 2015. MATCHUP: Memory Abstractions for Heap Manipulating Programs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '15)*. ACM, New York, NY, USA, 136–145. https://doi.org/10.1145/2684746.2689073

[118] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 845–858. https://doi.org/10.1145/3373376.3378491

[119] Yue Zha and Jing Li. 2021. Hetero-ViTAL: A Virtualization Stack for Heterogeneous FPGA Clusters. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 470–483. https://doi.org/10.1109/ISCA52012.2021.00044

[120] Yue Zha and Jing Li. 2021. When Application-Specific ISA Meets FPGAs: A Multi-Layer Virtualization Framework for Heterogeneous Cloud FPGAs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 123–134. https://doi.org/10.1145/3445814.3446699

[121] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '15)*. ACM, New York, NY, USA, 161–170. https://doi.org/10.1145/2684746.2689060

[122] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. 2016. Towards High Performance Paged Memory for GPUs. In *HPCA*.

[123] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. 2018. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)* (Feb 2018).