FlexiMem: Flexible Shared Virtual Memory for PCIe-attached FPGAs

Canberk Sönmez, Mohamed Shahawy, Cemalettin Cem Belentepe, Paolo Ienne
École Polytechnique Fédérale de Lausanne, CH-1015 Lausanne, Switzerland
{canberk.sonmez, mohamed.shahawy, cemalettin.belentepe, paolo.ienne}@epfl.ch

***Abstract*—Shared Virtual Memory (SVM) is a mechanism that allows host-side applications and FPGA accelerators to access the same virtual address space. It enables accelerating algorithms with unpredictable memory access patterns by making transparent pointer sharing possible. Even for applications with predictable memory access patterns, SVM helps by eliminating manual data movement. FlexiMem is a customizable SVM system that uses FPGA-addressable memory resources to store virtual memory pages, rather than directly accessing the host memory, to achieve high throughput and low latency. On the FPGA side, FlexiMem features a highly-flexible interconnect that, for the first time, allows for configuring address and payload data paths independently for SVM systems. It supports multiple master and slave devices sharing the same address space, such as accelerators issuing many memory requests in parallel to multiple memory banks. With our interconnect design, we provide numerous specialization dimensions to optimize the SVM system for a given workload. For example, irregular applications with short accesses to memory require an address translation for each data transfer. Such applications can take advantage of a highly parallelized address data path and an increased number of TLBs working in parallel. In contrast, regular and bursting applications transfer a small number of address packets per many data packets, and they can tolerate a lightweight address data path with a single translation unit. FlexiMem also provides address translation units with reconfigurable capacities and page sizes to be tailored to the needs of the application. On the host side, FlexiMem leverages the Linux *userfaultfd* API and vendor-provided IPs and drivers for automatic data movement initiated by software. Blocks of memory allocated by the FlexiMem API can be passed freely to the FPGA or other host-side libraries, without requiring any kind of explicit data movement. We evaluate several experimental setups with various FlexiMem configurations to showcase the effect of customizability on performance.**

***Index Terms*—Shared Virtual Memory, FPGA**

I. Introduction

The ubiquitous paradigm of temporal computation is fundamentally inefficient: software programmability wastes too much time and power on data movement and control rather than computation. Reconfigurable computing, currently embodied primarily in Field Programmable Gate Arrays (FPGAs), holds the promise of offering the flexibility of traditional processors with some of the efficiency of dedicated circuits [1]. All large public cloud providers (Amazon, Baidu, Microsoft, etc.) have recently added FPGAs to their offering while the cloud processor market leaders (Intel and AMD) have purchased the two major FPGAs vendors Altera and Xilinx.

In datacenters, FPGAs are attached to a host CPU, usually over PCIe. FPGA accommodates accelerators and the CPU runs a host-side application. In many cases, the host-side application (1) moves data the accelerator will operate on from the host memory to the FPGA (over PCIe, for example), and (2) sends tasks to the accelerator. Then, the accelerator issues read/write requests to FPGA-addressable memories, such as BRAM [2], HBM [3], or DRAM [4], to fetch data and store the results. Such a programming model is most suitable for applications with predictable memory access patterns: the host can transfer data as the computation progresses. For irregular applications that deal with sparse, linked, and hierarchical data structures, this programming model can still work if the dataset fits in FPGA-addressable memory. However, irregular applications (e.g., data analytics) with large datasets require a fundamentally different approach due to the unpredictability of the memory access pattern.

A similar problem has been studied in the context of general-purpose computing for many decades. *Virtual memory* (VM) proved to be a groundbreaking abstraction that, in addition to its other merits, enabled scaling applications well beyond the limits of the available physical memory [5]. Today, it is almost impossible to think of a serious computing system that does not provide hardware and operating system support for it. A VM system provides an application with a practically unbounded *virtual address space* that is backed by a large storage called *swap*. Application data is brought from the swap to the main memory at *page* granularity in case of *page faults*, for example, when the application accesses absent pages. Then, a *translation lookaside buffer* (TLB) performs virtual-to-physical address translation for accessing the page present in the main memory. CPU/FPGA heterogeneous systems might benefit from a similar idea, by utilizing the host memory as a type of swap. That is, pages could be brought to FPGA-addressable memories from the host memory when needed.

Shared Virtual Memory (SVM) is a form of VM where both the accelerator and the host-side application share the same virtual address space. SVM allows transparent pointer passing between the host-side application and the FPGA-side accelerator, thereby facilitating the development of CPU/FPGA heterogeneous systems that operate on sparse, linked, and hierarchical data structures. Even for regular workloads, SVM obviates the need for explicit data transfers, greatly improving the programmer experience.

Despite its unquestionable usefulness, today, there is no easy-to-adapt SVM solution to exploit high bandwidth memory resources present on FPGAs. Prior work either attach the

1946-1488/24/$31.00 ©2024 IEEE
DOI 10.1109/FPL64840.2024.00020

SVM mechanism to a single memory port, or use several memory ports for independent applications with separate address spaces. Such solutions fall short for applications that consist of many accelerators [6], each of which (1) sharing the same address space and (2) issuing simultaneous memory requests targeting a larger number of memory ports. Furthermore, they depend on frameworks distributed by vendors and cloud providers, making it difficult to deploy the design on a wide range setups.

In this paper, we present FlexiMem, a customizable SVM system for PCIe-based CPU/FPGA heterogeneous systems. FlexiMem precisely targets the problem that we mentioned earlier: it provides the necessary tools, both in hardware and software, to construct an easy-to-deploy SVM system that exploits the high-bandwidth memory resources. We embrace customizability, reuse of existing components, modularity, and ease of deployment as part of our design philosophy. As a consequence of this philosophy, we eventually plan to release FlexiMem as an open-source product.

On the FPGA side, FlexiMem features a customizable interconnect to connect accelerators to high-bandwidth memory. On one extreme, applications with regular and bursting accesses require few address translations per data transfer. While still retaining a large data bandwidth, we can reduce the address translation bandwidth of such applications to save FPGA resources. On the other extreme, irregular applications need to perform many address translations per data transfer and require a larger address bandwidth. Between these two extremes, there are many cases not captured by the traditional interconnect topologies. In light of this, the FlexiMem interconnect that, for the first time, allows for specializing address and data bandwidths independently, to suit a given workload.

FlexiMem also provides TLBs with configurable capacities and page sizes to better suit a given workload. FlexiMem relies on a universally-available set of vendor-provided components, namely PCIe and memory IPs, for implementing the SVM functionality; therefore, it can be easily deployed on different FPGA platforms. On the host side, FlexiMem leverages the vendor-provided PCIe drivers for page transfers and the *userfaultfd* API [7], for the first time in the context of SVM, to handle page faults. Such an approach obviates the need for writing specialized kernel drivers—potentially for each FPGA vendor—and improves the portability of FlexiMem.

II. Related Work

We can classify the prior SVM implementations under two categories: *direct* and *indirect*. Direct implementations feature a single copy of each page, stored in the host memory for PCIe-attached FPGAs. Having only a single copy avoids coherency problems. A *translation layer* translates virtual addresses (VA) into physical addresses (PA) in the host physical address space. Since virtual-to-physical mappings are inherited from the OS *page table*, direct implementations usually choose the same page size as the OS.

Well-known FPGA application frameworks for PCIe-attached FPGAs, such as Coyote [8] and XRT [12], support the direct flavor of SVM. Users of such frameworks are usually required to configure the translations manually. PCIe [13] and upcoming cache-coherent interconnect standards, namely CXL [14], CCIX [15], and CAPI [16], provide address translation services that might obviate the need for manual configuration; however, these standards are not widely adopted. In our work, we took inspiration from the design decisions and implementations of open-source direct SVM frameworks, especially Coyote [8]. However, these studies are not a drop-in replacement for our purposes because they do not utilize on-FPGA memory resources, such as HBM, for page storage.

Direct SVM is also employed by various SoC-based systems [11], [17]–[19]. In particular, Vogel et al. [18] focus on the design space exploration of *translation lookaside buffers* (TLBs), which store address mappings and perform translations. Their work presents a flexible, fully-associative, single-cycle L1 TLB with limited capacity, being supported by a multi-cycle, set-associative L2 TLB with larger capacity. The same group also improves the TLB fault handling—when TLB misses a translation—by implementing an accelerator-side thread that manages the virtual memory system autonomously [11]. The hardware thread performs *page table walks*, i.e., iterating the page table to retrieve missing TLB translations, on the host page tables. While these studies target direct SVM for SoCs, their insights into TLB design and page table walks are still applicable to our work.

Indirect implementations, on the other hand, copy pages from the host memory to on-FPGA memory, and accelerators work with these copies. VAs are translated into PAs in the FPGA physical address space. In case the requested VA is not present on FPGA, the corresponding page is transferred automatically. OS-side and FPGA-side translation mechanisms are distinct, which in turn provides more opportunity for customization. Compared to direct implementations, caching pages in low-latency and high-bandwidth memory resources—such as BRAM, DDR, or High Bandwidth Memory, HBM—might provide a superior performance than accessing the host memory over PCIe, which has a higher latency and limited bandwidth. However, since a given page has multiple copies, indirect implementations should handle coherency issues. Indirect SVM implementations can also take advantage of existing PCIe Direct Memory Access (DMA) IPs and/or drivers for transferring data between the host and the FPGA.

Kalkhof et al. [10] provide an indirect implementation for PCIe-attached FPGAs that leverages Heterogeneous Memory Management (HMM) API provided by the Linux kernel. While being similar to our work in spirit, HMM enforces a fixed page size of 4 KiB. It is worth noting that, to support small pages efficiently, the authors had to provide specialized IPs and drivers. The implementation features a two-level *translation lookaside buffer* (TLB) to store translations. At the time of writing, it works with Alveo U280 and without HBM [20]. Ng et al. [21] propose an indirect SVM system for PCIe-attached FPGAs. However, their work stores only 4 pages on the FPGA device, with a fixed page size.

FSRF [9], while not being a SVM implementation, provides

TABLE I: Comparison of FlexiMem with prior work.

Feature	Our work	Coyote [8]	FSRF [9]	Tapasco SVM [10]	Vogel et al., 2017 [11]
Shared Virtual Memory	Yes	Yes	No, VAs for files	Yes	Yes
Deployment	PCIe-attached	PCIe-attached	PCIe-attached	PCIe-attached	SoC
On-device page storage	Yes	No, direct SVM	Yes	Yes	No, direct SVM
Addressable memory limited by	Host virtual memory	Host virtual memory	Disk storage	Host virtual memory	Host virtual memory
Host/device coherency protocol	ESI	None, direct SVM	None, not SVM	Not mentioned	None, direct SVM
Automatic data transfer for host-side faults	Yes	Not needed	No, sync explicitly	Yes	Not needed
Available page sizes	Any page size	4 KiB, 2 MiB	4 KiB, 2 MiB	4 KiB	Host page size
Dependence on other frameworks	No	No	AWS runtime	No	No
Reuse vendor-provided drivers/IP	Optional	Yes	Yes	No	Unknown

a virtual memory system in an indirect fashion. Its virtual address space indexes locations on an I/O device, such as disk. It constructs a multi-level cache hierarchy that consists of on-FPGA DRAM, the host memory, and NVMe storage. FSRF hardware supports configurable page sizes; however, the current software implementation supports only 4 KiB and 2 MiB pages. FSRF proposes a high-capacity DRAM-based TLB to store translations. The high latency of the DRAM TLB is compensated by bursting, which enables using a single address translation for multiple beats of data [22]. However, FSRF still proposes an SRAM-based TLB for workloads that cannot utilize bursts, such as workloads with irregular memory access patterns. Since FSRF is not a SVM implementation, it requires manual synchronization to ensure coherency. Furthermore, it shares the available DRAM bandwidth among multiple applications with distinct address spaces rather than allocating the entire bandwidth to a single application with many accelerator sharing the same address space.

Qilin [23] explores the design space of SVM implementations. It provides an open-source SVM implementation as a test-bed for design space exploration. However, Qilin is a direct SVM implementation, designed for experimentation purposes and constrained by the host's VM configuration.

In Table I, we compare our work with the most relevant prior work. We observe that none of the prior work provides configurability, coherency with the host, and automatic data transfer on host-side faults in a single package.

III. Design

A FlexiMem system, whose physical components are shown in Figure 1, consists of multiple *agents* that share the same virtual address space. These agents include the host-side application and on-FPGA accelerators. Each agent accesses its own copies of pages, rather than accessing the host-memory. FlexiMem incorporates a 3-state coherency protocol (exclusive, shared, and invalid) to keep individual copies of pages in sync. Unlike prior work that enforces the same page size as the host [8], [10], [11], our design allows for choosing the page size that best suits the workload. We use the term "page size" for FlexiMem pages and "native page size" for the operating system pages.

On the host side, the CPU executes the host-side application and the FlexiMem library, which implements the shared virtual

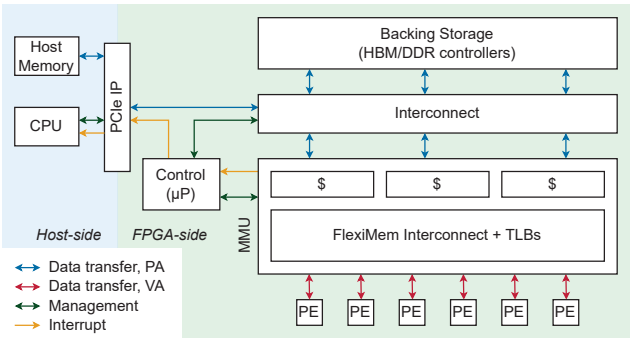

Fig. 1: FlexiMem architecture. CPU runs the host-side application and the FlexiMem library. Accelerators access the backing storage, rather than the host memory, for improved throughput and latency. Memory Management Unit (MMU) includes the FlexiMem interconnect, caches, and TLBs.

memory functionality. The available memory on the host side (including main memory and swap) is large enough to accommodate the application data entirely, which can reach TBs. The host-side application sends tasks to accelerators on the FPGA, and these tasks contain pointers to the virtual address space. Data and message transfer between the host and the FPGA takes place over PCIe, the standard connection protocol for datacenter FPGAs. On the FPGA-side, accelerators issue memory transactions with virtual addresses, and they access data stored in FPGA-addressable memories. Such an approach provides a higher throughput and lower latency compared to designs where accelerators directly access the host main memory. FlexiMem components perform virtual-to-physical address translations and page fault handling.

Figure 2 summarizes the deployment flow of the system. We provide RTL generators (written in Chisel3 HDL) that generate the FPGA-side components of a FlexiMem system based on a user-provided configuration file. The configuration file includes FlexiMem parameters, such as the page size, FlexiMem interconnect configuration, and the TLB configuration. These components can later be incorporated in a Vivado or Quartus project that also contains (1) the PCIe IP for data/message transfers and (2) a soft-core microprocessor

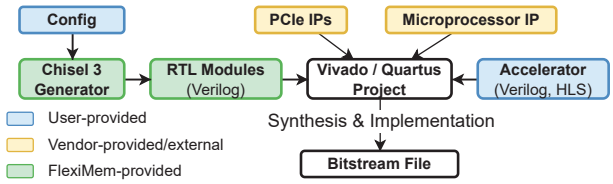

Fig. 2: FlexiMem deployment flow. Configuration file includes the page size, interconnect and TLB configurations.

for running the FlexiMem firmware. We also supply some helper scripts to automate the integration, for a few select FPGA models; however, it should be straightforward to adapt the system for any setup. After connecting the accelerator to FlexiMem modules, the system is ready to be deployed.

A. FPGA-side Architecture

FlexiMem consists of five components on the FPGA side. *Backing storage* keeps the pages and FlexiMem-related data structures (page tables, for example); it can be constructed using any kind of memory resource that the FPGA platform provides (e.g., DDR, HBM, or BRAM). *Memory Management Unit* (MMU) includes TLBs, which perform the virtual-to-physical address translation, and optionally, caches. MMU also includes the *FlexiMem interconnect* that connects PEs to TLBs, and TLBs to caches or the memory ports. *PCIe IP* performs DMA transfers to move pages between the host memory and the FPGA. It also provides interrupt functionality and memory-mapped register access to the hardware components from the host side. *Control unit*, which consists of a soft-core processor and a firmware, handles SVM-related tasks. Namely, it (1) handles TLB faults (i.e., missing TLB entries) by doing page table walks, (2) generates required coherency messages in case of page faults (i.e., missing or shared pages), and (3) cooperates with the host-side FlexiMem library for page fault handling. *Interconnect*, which directly connects to the backing storage, arbitrates the memory access requests coming from MMU, the PCIe IP, and the controller. All FPGA-side FlexiMem components communicate over AXI4, AXI4-Lite, and interrupt interfaces.

FlexiMem Interconnect: FlexiMem features an interconnect that, in contrast to traditional designs, features a highly customizable topology. Traditional interconnect designs usually employ either single-address multiple-data (SAMD) [24] or multiple-address multiple-data (MAMD) [25], [26] topology, both of which are shown in Figure 3. The first topology has a limited address bandwidth: it serializes address packets but allows simultaneous transfer of multiple data packets (also know as *data beats*). Such an approach is suitable for *bursting* workloads that send many data beats per address packet. However, it suffers from poor performance for irregular workloads that consist of single-beat transfers. The latter approach allows simultaneous address packet transfer by scaling the address bandwidth to match the data bandwidth; however, that comes at an exacerbated hardware cost. For an interconnect with 128 bits of data packets and 32 bits of address packets, the additional overhead would be approximately 25%—i.e., fraction of the address and data packet widths. For bursting workloads, such an overhead is not justified because the address bandwidth is never utilized as much as the data bandwidth. Even for single-beat workloads, address datapath tends to get underutilized because the target memory port suffers from a performance bottleneck due to lack of bursts. In light of these observations, we believe that the ideal interconnect topology for a SVM system lies between these two extremes and propose the FlexiMem interconnect.

For the first time, the FlexiMem interconnect allows specializing address and data bandwidths independently from each other, presenting a larger design space to better suit any given workload in terms of resource utilization and performance. It is an AXI4-based interconnect and capable of customizing each of the AXI channels (read address/data, write address/data/response) separately. Figure 3 shows a FlexiMem interconnect configuration with an address bandwidth of 2 packets/cycle and a read data bandwidth of 8 packets/cycle. Arbiters constitute the first stage of the address pipeline because the crossbar bandwidth is not sufficient to serve all the accelerators simultaneously. TLBs at the output of the crossbar perform address translation. The number of TLBs is the same as the number of crossbar ports to match the address bandwidth. The user can replace TLBs with custom implementations—featuring increased capacity and associativity, for example—thanks to the latency-insensitive protocol employed. After translation, address packets are routed to the correct memory bank via demultiplexers. It is possible to choose the address bits that index TLBs and memory banks, depending on the memory access pattern of the workload. The interconnect employs the ID field of the transaction to routing the responses (i.e., read data and write response) back to the accelerator.

We note that vendor-provided interconnect IPs [24]–[26] are not as customizable as the FlexiMem interconnect. It is not possible to modify their address decoding logic from outside to accommodate the address translation needs. Consequently, each memory bank, cache, or accelerator would need a separate TLB with such interconnects, irrespective of whether the workload could benefit from them. Finally, they cannot be easily instantiated from RTL code, resulting in fragmentation in the system design and difficult integration.

Page tables: In VM systems, *pages* in the virtual address space are mapped to regions in the physical address space called *page frames*. Pages and page frames are identified using *Virtual Page Numbers* (VPNs) and *Physical Page Numbers* (PPNs) respectively. The *page table* is a data structure that stores VPN-to-PPN mappings. It consists of *page table entries*, that contain status flags, VPN, and PPN. The control unit scans page table entries to retrieve VPN-to-PPN mappings in case of TLB faults, or modify it in case of page faults (i.e., when the page is not present or not in the correct state).

FPGA page table structure differs from OS page tables. OS page tables have virtually unlimited storage, as they are defined in terms of kernel virtual memory which also

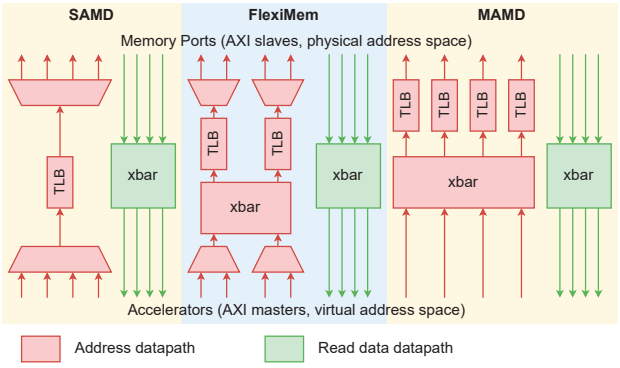

Fig. 3: Single-address multiple-data (SAMD), multiple-address multiple-data (MAMD) topologies and the FlexiMem interconnect (only the read channel). In the SAMD topology, the address bandwidth is limited to 1 address packet/cycle, whereas in the MAMD topology, the address bandwidth matches the data bandwidth. The FlexiMem interconnect has a customizable topology that covers the design space in between.

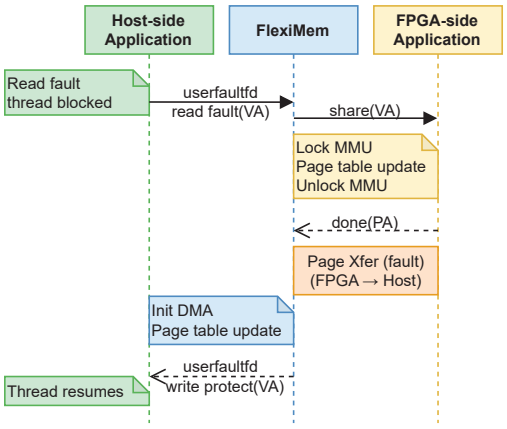

Fig. 4: An example sequence of events for host read faults.

extends to the swap area; therefore, they can—and must—store every mapping that ever existed [27]. In contrast, FPGA page tables are accommodated in the physical address space, so they have a limited number of page table entries. For this reason, we use a set-associative hash table for storing page table entries [28] rather than multi-level page tables commonly employed by modern operating systems [29]. Each page table entry repurposes the most significant 8 bits of the address—which are not used by the OS for userspace pages— for encoding the state information (shared, exclusive, or invalid). FPGA page tables grow until reaching a maximum size. It is possible to configure the associativity and the initial number of sets depending on the application requirements. One possible issue with our design is eviction of pages due to page table entry contention, as the number of entries within a set is limited. However, alternative hash table designs that use open addressing schemes (e.g., linear probing) fail to provide latency guarantees, which is essential for TLB fault handling. In addition, we need PPN-to-VPN mappings to implement the page eviction algorithms; hence, we also have a reverse page table, similar to the `rmap` data structure used in Linux.

Page Eviction: When a page needs to be brought to the FPGA memory, the controller needs to find (1) an available entry in the page table, (2) an unoccupied page frame. If there is no available entry, the controller needs to evict an existing entry, preferably a not-recently-used (NRU) or a shared one. If the evicted entry refers to an exclusive page, the page must be written back to the host memory. The page fault is served using the frame of the evicted entry. If there is an available entry, the controller continues with finding an unoccupied page frame. In case there is no unoccupied page frame, the controller evicts a page, that is preferably NRU or shared. The evicted page must be written back to the host memory if it is exclusive. The page fault is served using the frame of the evicted page. The page table entry and page eviction algorithms we described so far are not hard-coded in hardware, they are part of the firmware. The user can alter the page eviction algorithm if needed.

B. Host-side Architecture

The host executes the application thread and the FlexiMem library. The FlexiMem library keeps track of pages and forwards the coherency messages among the agents. It also initiates data transfers via the PCIe IP drivers.

The host-side application generates coherency messages. We leverage the *userfaultfd* API, for the first time in the context of SVM, to trap memory accesses that generate page faults on the host side. For example, when the FPGA requests a page for reading, we mark the page as write-protected on the host-side, so that subsequent write accesses are trapped by the *userfaultfd* monitoring thread and necessary coherency messages are sent to the host. If a page is owned exclusively by the FPGA, the page is marked as invalid on the host and any kind of access is trapped. This approach eliminates the need for manual user-initiated data transfers. It is worth noting that the *userfaultfd* API is available on any modern Linux kernel, and it does not require any form of modification to the kernel; therefore, it contributes significantly to the portability to FlexiMem. For initiating the data transfers, we take advantage of vendor-provided PCIe drivers. Using *userfaultfd* and vendor-provided PCIe drivers instead of writing specialized kernel drivers also saves the user from a significant amount of engineering work.

C. Coherency

FlexiMem employs a 3-state coherency protocol with Exclusive, Shared, and Invalid states (ESI) to keep host and device copies coherent. An agent can read from Exclusive or Shared pages, and write only to Exclusive pages. Agents generate and receive *coherency messages*, namely “Get”, “Get for writing”, “Upgrade for writing”, “Flush”, “Share”, and “Invalidate”. These coherency messages are generated in case of page faults, for example, when an agent writes to a page that it does not have; this creates a “Get for writing” message.

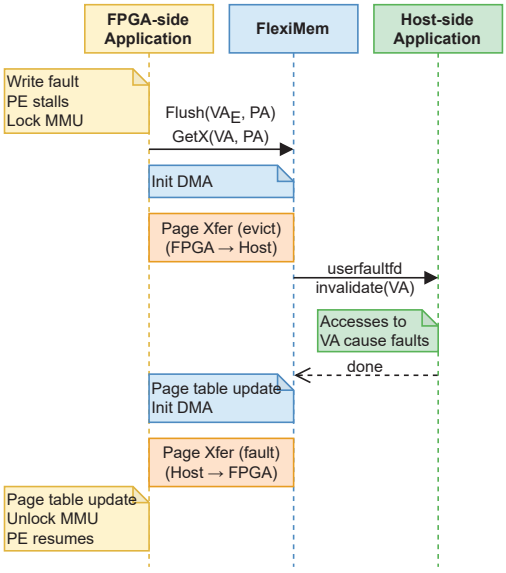

Fig. 5: An example sequence of events for FPGA write faults when there are no free page available on the FPGA side.

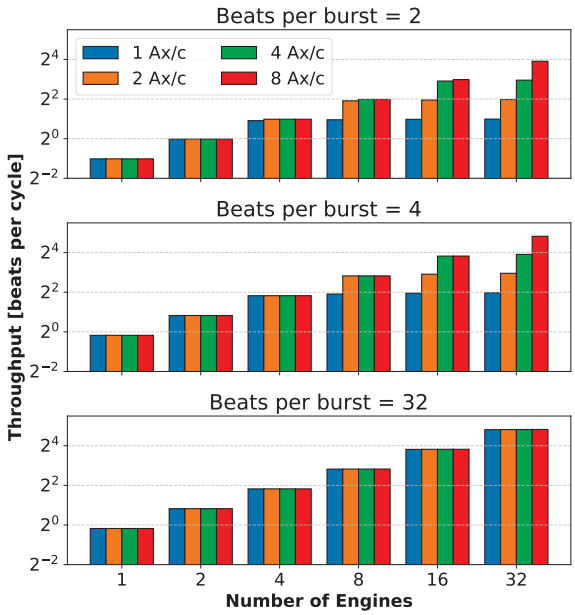

Fig. 6: Effect of the address bandwidth for various workload type characterized by the burst length. Ax/c stands for address packets translated per cycle (i.e., the number of TLBs), which is same for both read and write address channels. The optimal address bandwidth depends on the nature of the workload. The FlexiMem interconnect can be configured to provide exactly the address bandwidth at which the throughput saturates to avoid wasting resources.

We explain how coherency is handled by analyzing two scenarios: (1) a read page fault on the host side caused by an access to an exclusive page on the FPGA (Figure 4) and (2) a write page fault on the FPGA side caused by an access to an exclusive page on the host (Figure 5). In the latter scenario, we assume that the FPGA needs to evict a page. We ensure that MMUs are locked and all AXI transactions are complete before performing any page transfers to avoid data corruption.

Writes-in-flight: FlexiMem avoids data corruption by waiting for all outstanding read and write transactions are complete on the FPGA side. An AXI write transaction consists of an address packet and possibly many data packets. The transaction starts when an address packet leaves the MMU. If data packets of a write transaction depend on a read transaction stalling on a page fault, the system comes to a deadlock. For this reason, the FlexiMem MMU allows the address packet of a write to proceed only when all its data packets are available.

IV. Evaluation

A. FlexiMem Interconnect

In this section, we provide an evaluation of the FlexiMem interconnect, with configurations that span the design space from single-address multiple-data to multiple-address multiple-data topologies. We compare these different configurations from performance and resource utilization perspectives. We then argue that, more often than not, the optimal design point is not attainable by traditional interconnect topologies but the optimal is indeed covered by the design space provided by the FlexiMem interconnect. Our interconnect evaluations assume the common case of the SVM system, that is, no page or TLB faults. This is a valid assumption, as we later discuss, our system favors larger page sizes and data reuse.

Due to the large number of data points, we evaluate the performance of the FlexiMem interconnect in simulation. We use cycle accurate simulation models for the FlexiMem interconnect and accelerators, both of which are obtained from synthesizable RTL. The simulation environment is based on SystemC, we use Verilator to convert Chisel3-generated Verilog files to SystemC models. We employ a memory model based on the HBM IP of Xilinx Alveo U55C FPGA. Our simulated design uses Scatter-Gather DMA (SGDMA) engines as accelerators and they are connected to the interconnect. SGDMA engines can be programmed on the fly to create desired memory access patterns. We synthesize the design for Alveo U55C FPGA (VU47P) using Vivado 2022.2 for resource utilization comparison.

Figure 6 shows the system throughput (beats/cycle) for various types of workloads characterized by the burst length. SGDMA engines create memory requests non-stop to imitate a memory-bound algorithm—the most challenging type of algorithms for FlexiMem. We make sure that the interconnect is the only potential bottleneck in the system is by having a sufficient number of memory banks with no contention. We measure the system throughput in terms of beats per cycle, which is at most the same as the number of SGDMA engines.

For a small number of beats, the system bandwidth is

TABLE II: The hardware cost of a 32×32 FlexiMem interconnect, with 32-bit addresses and 128-bit data. Data bandwidth of the interconnect is fixed at 32 data beats/cycle and the address bandwidth is varied. The first row corresponds to the SAMD topology and the last row corresponds to the MAMD topology. Small increases in the address bandwidth do not change the hardware cost significantly. However, if it matches the data bandwidth, the LUT utilization increases by 45%.

	LUT	Util.	FF	Util.	BRAM	Util.
Available	**1,303,680**	**100.00%**	**2,607,360**	**100.00%**	**2,016**	**100.00%**
1 addr/cycle	122,582	9.40%	45,175	1.73%	0	0.00%
2 addr/cycle	122,530	9.40%	45,219	1.73%	0	0.00%
4 addr/cycle	123,545	9.48%	45,240	1.74%	0	0.00%
8 addr/cycle	125,181	9.60%	45,326	1.74%	0	0.00%
16 addr/cycle	135,811	10.42%	45,539	1.75%	0	0.00%
32 addr/cycle	178,367	13.68%	45,728	1.75%	0	0.00%

bottlenecked by both (1) the reduced efficiency of the memory and (2) contention for the limited address bandwidth. In this scenario, increasing the address bandwidth indeed helps improve the system throughput; therefore, the SAMD configuration is not optimal. However, a bandwidth of 8 addresses/cycle is sufficient to saturate the performance in all cases. An increased address translation capability does not improve the throughput further because the reduced efficiency of memory makes the memory bandwidth, and not the address translation, the bottleneck. In other words, MAMD with an address throughput of 32 address/cycle is an overkill. In fact, as shown in Table II, using MAMD in this scenario would waste an additional 45% of resources. Obviously, the ideal design point is neither SAMD or MAMD. Thanks to its customizable nature, the FlexiMem interconnect can provide the optimal solution as a function of the workload, memory parameters, and the number of accelerators.

For a larger number of beats, the system throughput shows no sensitivity to the address bandwidth, and scales perfectly with the number of engines. This indicates that the interconnect has a good RTL implementation. We finally note that the performance results we present in this section might change for different memory parameters, we can still find a suitable FlexiMem interconnect configuration.

B. Host-FPGA Interaction

In this section, we evaluate the performance of the FlexiMem system on a fully-integrated CPU/FPGA setup. In contrast to the previous section which focused on the common case of SVM, in this section, we direct our attention to the page fault case. We test our system on FPGAs provided by ETHZ Heterogeneous Accelerator Cloud Cluster (ETHZ-HACC) [8]. We used an Alveo U55C FPGA (VU47P), synthesized our design using Vivado 2022.2, as in the previous section. All the components work at 250 MHz, as dictated by the PCIe DMA IP. While we tested our system in the Xilinx ecosystem, there is nothing that fundamentally stops FlexiMem from being implemented on Altera FPGAs and we have plans to support them in future.

We aim to measure the impact of FlexiMem on the accelerator execution time as a function of the page size and the amount of the data consumed. On the FPGA side, our evaluation system contains fully-associative and combinational TLBs with 8 entries and various page sizes. At a given time, only one TLB is active. The accelerator applies an in-place transformation to the input data. On the host side, we use the SVM library as-is with no special modifications for testing. The tester software enables a single TLB at a time and sends tasks to the accelerator. In all our experiments, we only use Xilinx-provided PCIe DMA IP [30] and its open-source driver [31], both of which we integrated with FlexiMem easily.

The accelerator issues read requests to linearly-increasing addresses. It modifies the read data and writes the transformed data to the same address. Accelerator experiences two page faults per page. The first page fault happens when the data is read. The page is transferred to the FPGA in shared state. When the accelerator attempts to write to a shared page, another page fault happens. However, the second fault does not trigger a data transfer because the page is now present.

Figure 7 shows the system performance for various page and data transfer sizes, and compares it to a setup without SVM. In the plots, we refer to experiments using TLBs with the page sizes (32 KiB, 256 KiB, and 2 MiB), No-SVM experiment assumes data is already present on the FPGA, the execution does not involve any of the FlexiMem components. We measure the execution time of the accelerator to process some data of a given size (i.e., the x-axis). We observe that the performance of the FlexiMem system improves (i.e., shorter execution time) as the page size becomes larger as the number of page faults is reduced. Compared to the no-SVM case, the overhead goes down to 40% with a page size of 2 MiB. This is expected because a larger page size implies a smaller number of page faults. Note that our experiments test the worst case scenario for the FlexiMem system, i.e., no data reuse to amortize the SVM overhead. Such a scenario is extremely unlikely for workloads requiring SVM. More likely workloads with higher degrees of data reuse would experience a fewer page faults and a reduced SVM overhead.

To better understand the performance limits of a FlexiMem system, we measured the performance with userfaultfd disabled. In this scenario, there is no implicit data movement; the programmer needs to explicitly request pages from the FPGA to the host before accessing. Figure 8 plots the execution times with and without *userfaultfd*, with various page sizes, and reports the performance overhead of *userfaultfd* in percentage. We observe that the performance overhead of *userfaultfd* client increases with the page size. The overhead is around 10.8% for 32 KiB pages, 20.0% for 256 KiB pages, and around 46.9% for 2 MiB pages. The reason is that the *userfaultfd* API needs to mark native pages that correspond to FlexiMem page as invalid or read-only. The number of native pages to modify increases linearly as the page size increases. Therefore, while increasing

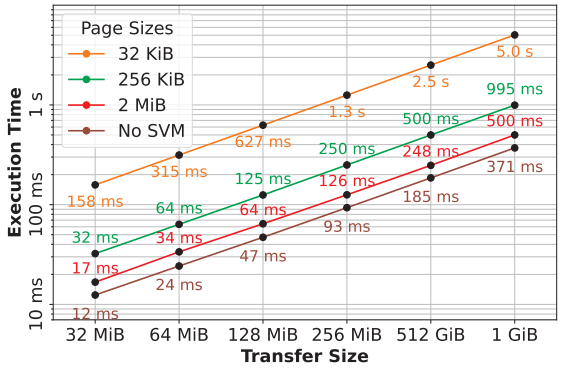

Fig. 7: Execution time vs. transfer size for various TLB sizes and without SVM. If pages are large enough, the overhead becomes more tolerable. Execution time is, compared to the no-SVM case, 13.3× worse for 32 KiB pages, 2.6× worse for 256 KiB pages, and only 1.4× worse for 2 MiB pages.

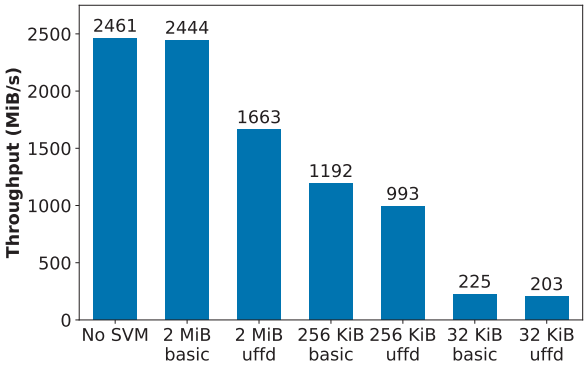

Fig. 8: Comparison of the system throughput with *userfaultfd* and without it (labeled "basic"). The performance overhead of *userfaultfd* increases with the page size because more native pages need to be marked invalid or read-only. FlexiMem system without userfaultfd reaches the no-SVM throughput with 2 MiB pages.

the page size increases the performance overall (as shown in Figure 7), *userfaultfd*'s contribution to the performance overhead increases linearly. To reduce the overhead, the user might either (1) avoid *userfaultfd* or (2) use larger native pages to decrease the processing time of the *userfaultfd* API. Please note that with the basic client, FlexiMem can achieve no-SVM throughput with 2 MiB pages.

The performance metrics we presented in this section are dominated by the page fault handling mechanism. The overhead mainly comes from (1) the userfaultfd API for large page sizes and (2) vendor-provided IPs and drivers for small page sizes. Of course, one can put an immense engineering effort into developing custom—and more efficient—PCIe IPs and kernel drivers [10], possibly for each and every FPGA model,

TABLE III: Resource utilization of typical FlexiMem components (except interconnect). Most components have less than 1% utilization. PCIe IP has significantly higher resource utilization; however, it needs to be present even without SVM. TLBs are reported with FlexiMem page sizes.

	LUT	Util.	FF	Util.	BRAM	Util.
Available	**1,303,680**	**100%**	**2,607,360**	**100%**	**2,016**	**100%**
TLB, 32 KiB	1,464	0.112%	1,102	0.042%	0	0.000%
TLB, 256 KiB	1,419	0.109%	1,054	0.040%	0	0.000%
TLB, 2 MiB	1,344	0.103%	1,006	0.039%	0	0.000%
Microblaze	3,698	0.284%	3708	0.142%	34	1.687%
HBM IP	778	0.060%	824	0.032%	2	0.099%
PCIe IP	5,4419	4.174%	54,089	2.074%	76	3.770%

to improve the performance metrics presented in this section. However, page faults are not the common case of an SVM system. More often than not, algorithms are tuned to exhibit greater degrees of locality, so that there are fewer and page faults. For this reason, we prioritize adaptability over the best imaginable page fault handling performance. We believe that our FlexiMem system provides an easy-to-adapt solution by relying only on vendor-provided IPs/drivers and the userfaultfd API. Nevertheless, the performance overhead of the FlexiMem system is basically negligible without the userfaultfd API for large page sizes. In other words, users of FlexiMem can trade off programmer comfort for performance by disabling the userfaultfd and automatic page transfers initiated by the host. This would not affect automatic page transfers initiated by FPGA-side page faults.

Table III lists the resource utilization of the most important components to build the FlexiMem system. Clearly, most of the FlexiMem building blocks have negligible utilization. Only the PCIe IP [30] suffers from a significant LUT and FF utilization (around 4% and 2% respectively); however, it is an essential component used even by non-SVM FPGA designs.

V. Conclusion

FlexiMem presents a modular, open-source, and portable SVM framework for CPU/FPGA heterogeneous systems. To accommodate the hardware acceleration of different classes of applications, it provides a variety of possible configurations. It extends the FPGA memory resources by employing the host memory as a type of swap for the FPGA-addressable memory. FlexiMem yields significant benefits for both unpredictable and predictable memory accesses. Unpredictable memory accesses are enhanced through the implementation of transparent pointer sharing, while predictable accesses are simplified by eliminating manual data access procedures. The evaluation of our design across varying data sizes and page sizes demonstrates that there is only a minimal overhead compared to other systems, while the effort required to create a full-featured SVM system is considerably reduced.

REFERENCES

[1] W. Najjar and P. Ienne, "Reconfigurable computing," *Micro, IEEE*, vol. 34, pp. 4–6, 01 2014.

[2] "Ip processor block ram (bram) block (v1.00a) data sheet(ds444)," https://docs.xilinx.com/v/u/en-US/bram_block, 2011, accessed: October 12, 2023.

[3] "Axi high bandwidth memory controller logicore ip product guide (pg276)," https://docs.xilinx.com/r/en-US/pg276-axi-hbm, 2022, accessed: October 12, 2023.

[4] "Ultrascale architecture-based fpgas memory ip v1.4," https://docs.xilinx.com/v/u/en-US/pg150-ultrascale-memory-ip, 2022, accessed: October 8, 2023.

[5] P. J. Denning, "Before memory was virtual," 1997.

[6] M. Shahawy, C. Sönmez, C. Belentepe, and P. Ienne, "Hardcilk: Cilk-like task parallelism for fpgas," in *32nd IEEE International Symposium On Field-Programmable Custom Computing Machines (FCCM'24)*. IEEE, May 2024, pp. 140–150.

[7] "Userfaultfd (the linux kernel documentation)," https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt, [n.d.], accessed: October 8, 2023.

[8] D. Korolija, T. Roscoe, and G. Alonso, "Do OS abstractions make sense on FPGAs?" in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 991–1010. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/roscoe

[9] J. Landgraf, M. Giordano, E. Yoon, and C. J. Rossbach, "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*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 556–571. [Online]. Available: https://doi.org/10.1145/3582016.3582048

[10] T. Kalkhof and A. Koch, "Efficient Physical Page Migrations in Shared Virtual Memory Reconfigurable Computing Systems," in *2021 International Conference on Field-Programmable Technology (ICFPT)*, Dec. 2021, pp. 1–10.

[11] P. Vogel, A. Kurth, J. Weinbuch, A. Marongiu, and L. Benini, "Efficient Virtual Memory Sharing via On-Accelerator Page Table Walking in Heterogeneous Embedded SoCs," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5s, pp. 154:1–154:19, Sep. 2017. [Online]. Available: https://doi.org/10.1145/3126560

[12] "Xilinx runtime library xrt," https://www.xilinx.com/products/design-tools/vitis/xrt.html, 2018, accessed: October 8, 2023.

[13] J. Ajanovic, "Pci express*(pcie*) 3.0 accelerator features," *Intel Corporation*, vol. 10, pp. 2–2, 2008.

[14] *Compute Express Link*, Compute Express Link Consortium, August 2023, rev. 3.1.

[15] G. Caruk, H. Abdulhamid, H. Ahn, J. Dastidar, K. Puranik, M. Mittal, and P. Mannava, "Cache coherent interconnect for accelerators: Ccix base specification," 2022, rev 2.0.

[16] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, "CAPI: A Coherent Accelerator Processor Interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7:1–7:7, Jan. 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7029171

[17] E. Lübbers and M. Platzner, "Reconos: Multithreaded programming for reconfigurable computers," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 1, oct 2009. [Online]. Available: https://doi.org/10.1145/1596532.1596540

[18] P. Vogel, A. Marongiu, and L. Benini, "Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU," *IEEE Transactions on Computers*, vol. 68, no. 4, pp. 510–525, Apr. 2019.

[19] F. Winterstein and G. Constantinides, "Pass a pointer: Exploring shared virtual memory abstractions in OpenCL tools for FPGAs," in *2017 International Conference on Field Programmable Technology (ICFPT)*, Dec. 2017, pp. 104–111.

[20] "Tapasco svm," https://github.com/esa-tu-darmstadt/tapasco/blob/master/documentation/tapasco-svm.md, accessed: October 8, 2023.

[21] H.-C. Ng, Y.-M. Choi, and H. K.-H. So, "Direct virtual memory access from FPGA for high-productivity heterogeneous computing," in *2013 International Conference on Field-Programmable Technology (FPT)*, Dec. 2013, pp. 458–461.

[22] *AMBA AXI Protocol Specification*, Arm Limited, September 2023, issue K.

[23] E. Richter and D. Chen, "Qilin: Enabling performance analysis and optimization of shared-virtual memory systems with fpga accelerators," in *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2022, pp. 1–9.

[24] ARM, "Logicore ip axi interconnect (v1.06.a) data sheet (axi)(ds768)," https://docs.amd.com/v/u/en-US/ds768_axi_interconnect, 2012, accessed: Mar 29, 2024.

[25] ——, "Smartconnect v1.0 logicore ip product guide [smartconnect (pg247)]," https://docs.amd.com/r/en-US/pg247-smartconnect/SmartConnect-v1.0-LogiCORE-IP-Product-Guide, 2022, accessed: Mar 29, 2024.

[26] A. Kurth, W. Rönninger, T. Benz, M. Cavalcante, F. Schuiki, F. Zaruba, and L. Benini, "An open-source platform for high-performance non-coherent on-chip communication," *IEEE Transactions on Computers*, vol. 71, no. 8, pp. 1794–1809, 2022.

[27] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating systems: Three easy pieces*. Arpaci-Dusseau Books, LLC, 2018, ch. 20.

[28] I. Yaniv and D. Tsafrir, "Hash, don't cache (the page table)," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, ser. SIGMETRICS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 337–350. [Online]. Available: https://doi.org/10.1145/2896377.2901456

[29] "Page table management," https://www.kernel.org/doc/gorman/html/understand/understand006.html, 2003, accessed: October 13, 2023.

[30] "Dma/bridge subsystem for pci express product guide pg195," https://docs.xilinx.com/r/en-US/pg195-pcie-dma, 2016, accessed: October 8, 2023.

[31] "Xilinx dma ip reference drivers," https://github.com/Xilinx/dma_ip_drivers/tree/master/XDMA/linux-kernel, [n.d.], accessed: October 13, 2023.