Efficient Physical Page Migrations in Shared Virtual Memory Reconfigurable Computing Systems

Torben Kalkhof, Andreas Koch
Embedded Systems and Applications Group
TU Darmstadt
Darmstadt, Germany
{kalkhof, koch}@esa.tu-darmstadt.de

***Abstract*—Shared Virtual Memory (SVM) can considerably simplify the application development for FPGA-accelerated computers, as it allows the seamless passing of virtually addressed pointers across the hardware/software boundary. Especially applications operating on complex pointer-based data structures can profit from this approach, as SVM can often avoid having to copy the entire data to FPGA memory, while performing pointer relocations in the process.**

Many FPGA-accelerated computers, especially in a data center setting, employ PCIe-attached boards that have FPGA-local memory in the form of on-chip HBM or on-board DRAM. Accesses to this local memory are much faster than going to the host memory via PCIe. Thus, even in the presence of SVM, it is desirable to be able to move the physical memory pages holding frequently accessed data closest to the compute unit that is operating on them. This capability is called physical page migration.

The main contribution of this work is an open-source framework which provides SVM with physical page migration capabilities to PCIe-attached FPGA cards. We benchmark both fully automatic on-demand and user-managed explicit migration modes, and show that for suitable use-cases, the performance of migrations cannot just match that of conventional DMA copy-based accelerator operations, but may even exceed it by overlapping computations and migrations.

***Index Terms*—shared virtual memory, SVM, page migrations, FPGA accelerators**

I. Introduction

Hennessy and Patterson identify domain-specific approaches in their 2018 Turing Award lecture [1] as one of the key enablers for future advances in computer architecture. Reconfigurable computers are, by their very nature, an ideal means to that end. However, the resulting need for communication between general-purpose and domain-specific processing elements raises new architecture challenges. One of these is the more complicated programming required when pointer-based data structures must be handled.

A proven approach to simplify this scenario is the use of Shared Virtual Memory (SVM) between processing units to allow the free exchange of pointers in a common address space. However, SVM alone does not fully solve the problem, as many processing elements have faster *locally* attached memories, leading to a NUMA systems architecture.

This research was funded by the German Federal Ministry for Education and Research (BMBF) with the funding ID 01 IS 21007 B.

Here, it is desirable to *migrate* the underlying memory pages with their data to physical memory *close* to the processing elements that currently work most intensively on the data. Thus, allowing those elements the fastest access, but still keeping the memory easily accessible for other processing elements that access it less frequently.

Our contributions to achieve such a behavior are threefold: (1) an open-source framework for easily building heterogeneous computing systems using shared virtual memory between CPU(s) and FPGA-based processing elements, (2) supporting efficiently migrating physical memory pages between the locally attached memories, and (3) benchmarking multiple operating modes suitable for different classes of applications.

We begin by introducing terminology and related work in Section II, present our systems architecture in Section III, which we then evaluate in Section IV. We conclude in Section V and look ahead towards future work.

II. Related Work

In conventional memory management schemes, such as the existing TaPaSCo framework [2], the hardware accelerator on the FPGA uses physical addresses to access data in memory. In the SVM concept, however, the accelerator operates in the same virtual address space as the user software running on the CPU. In modern operating systems, the mappings between virtual and physical memory pages are saved in multi-level page tables. Translating a virtual to a physical address by iterating over the page table is called a Page Table Walk (PTW). PTWs can be performed in software, or directly in hardware by a Memory Management Unit (MMU). MMUs also include Translation Lookaside Buffers (TLBs), which serve as a cache for recent address translations. One technique to implement SVM in a heterogeneous system with several memories are physical page migrations. Here, the memory pages are migrated to the memory, in which the compute unit requiring the data can access it most efficiently.

The majority of existing implementations of SVM for FPGA platforms use systems with a coherent or shared *physical* memory, which simplifies the task of keeping data consistent considerably. Embedded Systems-on-Chip (SoCs), such as the Xilinx Zynq [3] or Intel Cyclone [4] platforms, are thus quite suitable for SVM, since FPGA and CPU share the actual physical memory anyway.

Lange et al. [5] implement SVM on the ML310 reconfigurable SoC [6]. They use an IOMMU on the FPGA for address translations of the accelerator's memory requests, which contains a TLB, and is able to perform PTWs over the CPU page tables for address translations. Hence, only page faults must be handled in software on the CPU. To guarantee memory coherency, the CPU cache is flushed before launching the hardware accelerator. Furthermore, Lange et al. extend the OS kernel in order that TLB flushes are forwarded to the IOMMU as well.

Pass A Pointer [7], which implements support for OpenCL *Fine-grained system SVM* [8] on the Cyclone V platform [4], has a very similar approach to [5]. Additionally, the IOMMU of *Pass A Pointer* coalesces multiple faults on contiguous pages into a single request to reduce the overall number of interrupts. *Pass A Pointer* uses the Accelerator Coherency Port (ACP) to achieve cache-coherency at any time, and the IOMMU supports atomic memory operations from the FPGA and CPU.

Vogel et al. [9] propose a general SVM framework for embedded SoCs. Similar to our work, Vogel et al. do not include PTW support in their on-FPGA IOMMU, but handle TLB misses like page faults in software. They propose a two-level TLB hierarchy with a small Level 1 TLB with mappings of arbitrary length, and a large Level 2 TLB with a BRAM-based design. Compared to other on-FPGA IOMMU implementations, this approach allows much higher capacity at the cost of a variable look-up latency. However, the accelerator must be aware that the IOMMU returns error responses after TLB misses, and offers so-called *prefetching* commands to issue TLB misses in advance for more efficient fault handling. Also, the user must decide whether the coherent ACP or the more performant direct port to host memory is used for memory requests. This gives a programmer more flexibility and space for optimizations, but also requires more knowledge and intervention. In contrast, our framework does not require changes to most accelerators, and almost no guidance by the user software.

The Convey Hybrid-core machine [10] has two distinct memory pools attached to the host processor and co-processor respectively [11]. In contrast to our work, host and co-processor can access both memory pools, and different page sizes are supported. Nevertheless, the programmer is encouraged to explicitly place or migrate data to the closest memory pool by using compiler directives to achieve shorter access times. Unlike our framework, the Convey machine does not support automatic page migrations, though.

The work of Ng et al. [12] and IBM CAPI [13] target FPGA accelerator cards, and are both implemented on-top of PCIe, similar to our work. However, in the work of Ng et al. and in CAPI, the FPGA accesses data directly in host memory using PCIe, which leads to potentially higher latency and lower bandwidth than when using device local memory. Ng et al. simply pin required memory pages in host memory and flush the CPU cache before granting the hardware accelerator on the FPGA access to the data. Hence, they cannot guarantee data coherency for concurrent memory access from the CPU and FPGA. In CAPI, the FPGA shares system memory coherently with the CPU cores using the symmetric multi-processor (SMP) bus interconnect fabric to participate in the CPU core's coherency protocol. However, this requires dedicated hardware support on the host side, and thus, CAPI is only compatible with the POWER architecture. In order to increase performance, Ng et al. and CAPI provide a MMU with a TLB, as well as a data cache on the FPGA side. In CAPI the MMU includes additional PTW support.

Two further platforms with an even closer coupling between host CPU and FPGA are the Intel HARP platform [14] and *Enzian* [15]. While *Enzian* uses its custom coherent interconnect ECI [16], newer versions of HARP connect the FPGA cache-coherently via UPI to the distributed shared memory of the Xeon processor cores. Two PCIe links increase the maximum memory bandwidth further [17]. By using coherent connections to the CPUs, both platforms avoid again explicit data movements and ensure data consistency, however achieve higher bandwidth and lower latency than CAPI via PCIe [18].

Upcoming bus standards such as OpenCAPI [19], CXL [20] and CCIX [21] are designed to extend or replace PCIe and support cache-coherent accesses, virtual addressing, and the inclusion of device local memory in a system memory map. Nonetheless, PCIe-based accelerator cards are still widely-used.

In the GPU world, SVM has been used for a number of years now. It has been included in CUDA 6 (as *Unified Memory*) [22], OpenCL 2.0 [8] and HSA [23]. One main benefit of GPUs is the huge memory bandwidth. However, it can only be exploited if the data is actually located *in* GPU memory. As a solution all required data must be copied or migrated from host to GPU memory before starting the kernel execution. When using SVM, this is done with physical page migrations. Modern GPUs have advanced MMUs which on their own are able to trigger page faults for missing memory pages. The device driver is then responsible for migrating the required pages to GPU memory, and also back to host memory again after a CPU page fault has occurred. We adopt this technique of *Demand Paging* in our framework for FPGAs. While NVIDIA uses the generic NUMA migration functions in its proprietary driver [24], we use the Linux Heterogeneous Memory Management (HMM) API [25], similar to the Nouveau [26], AMD [27] and Intel drivers [28].

Current research in the field of SVM and page migrations for GPUs focuses on advanced topics such as enhanced hardware support [29], page placement in multi-GPU systems [30], [31], different page sizes [32], partial page migrations [33], and memory oversubscription and page eviction [34], [35].

Coyote [36] implements a basic version of page migrations as an OS abstraction for FPGAs. However, the implementation is solely for evaluation purposes and neither ensures data consistency during concurrent accesses, nor supports automatic back-migrations. Instead of using the Linux HMM API, *Coyote* simply pins the corresponding memory page in host memory, and copies it to device memory [37].

III. System Architecture

Our framework implements SVM support with physical on-demand page migrations for the common system architecture of a host CPU and a PCIe-attached FPGA accelerator card with on-device memory, such as our evaluation platform the VC709 [38]. Since the hardware accelerator operates on virtual addresses as well, we provide a device IOMMU on the FPGA itself, which takes over translating virtual to physical memory addresses, and is able to trigger page faults on the host. A device driver running on the host CPU is responsible for migrating memory pages to on-device memory after page faults, and managing the device IOMMU's TLBs. Also, the device driver needs to handle CPU page faults on migrated pages, and moves them back to host memory.

A. Building Blocks for Shared Virtual Memory

While the actual address translations are handled in hardware by the device IOMMU, the device driver running on the host CPU performs SVM management functions. Due to the growing importance of SVM, Linux provides platform and device support on the OS side in the form of the Heterogeneous Memory Management (HMM) API [25]. Its basic idea is to achieve a uniform view on all memories in a heterogeneous system. For this purpose, the API provides generic helper functions for device drivers. However, device specific tasks, such as device memory management, data transfers or TLB management, always need custom handling by the device driver itself.

One feature of HMM are *MMU notifiers*. A driver can use a MMU notifier to receive updates on the CPU page table of a process to manage on-device TLBs, or maintain its own page tables. We use this feature to invalidate TLB entries of memory pages which are freed on the host side while being physically located in device memory.

Furthermore, the HMM API supports page migrations to memories other than the host memory, which is the more important feature for us. Table I gives an overview of the used function calls. A device driver can allocate the usual `struct pages` in the common address space to represent its device memory, but has to remap them as device private pages. During the allocation the driver also has to register functions to handle a back-migration to host memory. As the first migration step, the driver uses the HMM API to perform a PTW collecting the required memory pages. The driver then allocates the device memory region, copies the actual data, usually using an DMA engine, and adds TLB entries to the device IOMMU. In case all TLB entries are in use, our IOMMU decides which entries get replaced by applying a pseudo-LRU strategy. Afterwards the HMM API updates the CPU page table so that it points to the `struct pages` representing the destination pages in device memory.

When the CPU tries to access a migrated page, the Linux kernel knows, using the device private entries in the page table, where the page is located at the moment. The page fault handler then calls the registered functions of the device driver to migrate the requested page back to host memory.

TABLE I
Relevant kernel and HMM API function calls for page migrations.

Function call	Description
`request_free_mem_region`	Allocate memory region for device private `struct pages`
`memremap_pages`	Remap pages as device private and register functions for back-migration
`migrate_vma_setup`	Perform PTW to collect pages to be migrated
`migrate_vma_pages`	Migrate `struct page` information from source to destination pages
`migrate_vma_finalize`	Update CPU page table

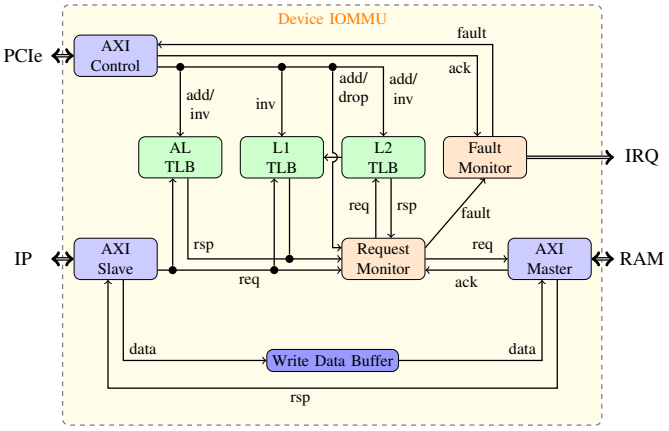


Fig. 1. Internal structure and data flow of the on-FPGA IOMMU. The abbreviations used in the figure stand for request (req), response (rsp), add entry (add), invalidate entry (inv), drop fault (drop), acknowledge (ack), arbitrary length TLB (AL TLB), and interrupt request (IRQ).

B. Implementation

In the following, we will briefly describe implementation details of our device IOMMU, device driver, and the feature of user managed page migrations.

1) Memory Management Unit: Figure 1 shows the internal structure of our on-FPGA IOMMU. It is written in Bluespec SystemVerilog (BSV) and designed for a maximum frequency of 200 MHz on the VC709 board [38]. We use the AXI4 standard [39] for all interfaces to ensure compatibility with many accelerators and other peripheral components. The device IOMMU handles up to 16 read and 16 write requests in-flight, and tracks the state of every memory request until its completion in a central *Request Monitor*. Every translation response by one of the TLBs, or new entries added by the driver, are compared with all pending requests and forwarded immediately to avoid redundant translations. Also, the device driver is able to check for active memory requests to a specific page at any time before triggering a back-migration to prevent potential data corruption or loss.

The device IOMMU uses a two-level TLB hierarchy. The Level 1 TLB is a fully-associative design, and realized with

LUTs and Registers. It translates one request per cycle, which leads to better performance in scenarios with many short bursts of requests addressing the same memory pages. As replacement strategy, we use a simple random replacement in case that all TLB entries are in use.

Invoking the driver after each TLB miss would be very costly. Our evaluation shows a 70 % higher runtime in a scenario where TLB entries get evicted (see Section IV-B). Hence, the Level 2 TLB focuses on providing high capacity. Here, we adopt the BRAM-based design from the work of Vogel et al. [9], but with added pipelining. This increases the TLB's maximum frequency to match our design frequency of 200 MHz and avoids the need for clock domain-crossing logic. Saving virtual and physical addresses in BRAMs leads to variable look-up times and lower throughput, since the BRAMs have to be searched *sequentially* for a matching entry due to the limited number of access ports. Also, the TLB cannot accept a new translation request, until the search for an entry of the previous request is finished. However, spreading the sets of virtual addresses over multiple BRAMs allows to recover some parallelism. In many applications, though, the limited throughput has no impact, since burst accesses to memory, allowing multiple data transfers with a single request, are used anyway. In contrast to the Level 1 TLB, the Level 2 TLB uses a pseudo-LRU replacement strategy. The TLB tracks which BRAM has been used least recently in the respective set, but evicts a random entry within this BRAM.

If neither TLB contains a matching entry, the device IOMMU raises an interrupt on the host to signal a page fault. It gathers up to 16 page faults in the *Fault Monitor*. In the case of concurrent faults on the same memory page, the *Fault Monitor* already removes the duplicates, and hands the respective page fault only once to the driver. After successful page fault handling, the driver adds new TLB entries directly to the Level 2 TLB. The Level 1 TLB is solely filled with hits from the Level 2 TLB.

2) Device Driver: The device driver is the central hub of the framework. It is the interface between the user space program, the OS kernel, and the hardware accelerator. It manages the device memory allocations and page migrations. Parts of the driver are inspired by the Nouveau driver [26] and the HMM test module [40] in the Linux kernel, since the use of the HMM API [25] imposes a basic common structure.

The device driver allocates the required `struct page` entries in chunks of 128 MB. It saves the physical base address of the chunk in device memory to be able to compute the actual page addresses based on chunk base address and page offset within the chunk later. A linked list contains all free pages and the driver allocates pages always in ascending order. In this manner, virtually contiguous buffers can be placed in physically contiguous memory as well, which allows some optimizations for migrations of large buffers. If a page is freed, the driver simply enqueues it again to the end of the linked list. Of course, TLB mappings for this page must be invalidated as well.

Similar to Vogel et al. [9], we handle page faults by the device IOMMU after an interrupt in a worker thread of the concurrency managed workqueue [41]. Figure 2 contains a flow diagram of the fault handling process. Like the IOMMU, the driver handles up to 16 faults in-flight. First, it checks with a PTW whether the pages are already present in device memory, and only the TLB mappings got evicted, since the device IOMMU has no means to differentiate between missing TLB entries and pages. As stated in Section V, future extensions to the IOMMU could include means to perform address translations without invoking the driver. Next, the driver sorts the faulting addresses to be able to coalesce adjacent pages in the following migration steps. It collects the pages to migrate with help of the HMM API [25], allocates pages in device memory, and transfers the actual data. As a DMA engine, we use our custom *PageDMA* core, which has a very simple structure, since it only copies or clears entire pages with a single AXI burst transfer each. Afterwards, the driver adds respective TLB entries to the IOMMU, and again uses the HMM API to update the CPU page table with the entries of the destination pages in device memory. It might occur that a CPU thread and the device try to access the same zero or as-yet unallocated page concurrently, and the migration fails [25]. In this case, the driver tries to migrate the page a second time, which is usually successful then. If the second attempt fails nonetheless, the fault is dropped, and the device IOMMU

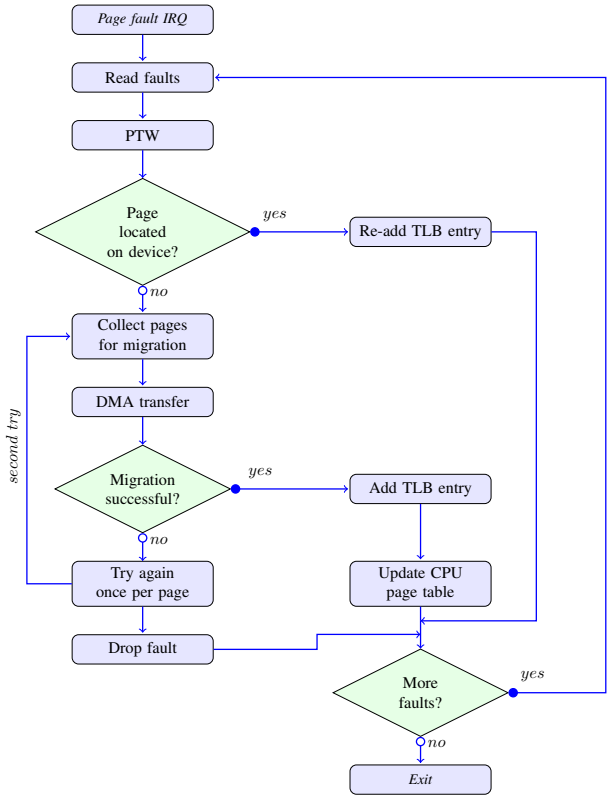


Fig. 2. Flow diagram of the worker thread handling device page faults. The thread is scheduled after an interrupt by the device IOMMU, and stays alive as long as further faults are pending. If the migration of a page fails, the thread takes a second try before dropping the fault.

returns an error response to the accelerator. After finishing handling the current page faults, the thread checks for new pending faults in the device IOMMU before exiting. In this manner, we reduce the number of interrupts, and avoid the overhead of immediately re-scheduling a new worker thread just after exiting the prior one.

Although the migration steps stay the same, the situation after a CPU page fault is slightly different. In this case, the Linux kernel handles the page fault, and only calls the migration functions the driver has to provide. Due to this, the driver has no chance to coalesce multiple faults, or postpone handling a fault. Every CPU page fault, and the resulting back-migration of the memory page, must be handled separately, and should be handled immediately, since the fault blocks the user thread. Additionally, the driver has to invalidate the corresponding TLB mappings in the IOMMU, and potentially wait for in-flight device-side accesses to the respective page to complete. Otherwise, the data of an in-flight write to a migrated-away page would get lost, and an in-flight read would incorrectly retrieve data from the *new* virtual page which replaced the formerly migrated page in this page frame.

3) User-Managed Page Migrations: Working with buffers in the range of many megabytes requires thousands of pages to be migrated to device memory and back. Doing this just with page fault triggered on-demand migrations can be very inefficient, especially the one-by-one migration back to host memory. Hence, we offer directives to the programmer to explicitly trigger the efficient migration of a memory region to or from device memory in advance, and not rely just on on-demand migrations. The device driver is able to significantly decrease the migration overhead by coalescing the migration of all pages containing the buffer.

Furthermore, the device IOMMU contains a third 8-entry TLB in parallel to the other two TLBs. It holds mappings with a flexible length of up to 4096 contiguous pages, similar to the Level 1 TLB by Vogel et al. [9]. Since our allocation system maps virtually contiguous buffers to physically contiguous pages, a single TLB entry may suffice to map an *entire* buffer. However, this type of TLB needs much more hardware resources, as simple comparators like in our other TLBs are not sufficient to find a matching entry. In multiple steps, the TLB calculates the offset of a request to every entry, checks afterwards whether the request is in the range of any mapping, and finally calculates the physical address based on this offset as well. Hence, it does not pay off to use this TLB for mappings with a length of less than or equal to 16 pages during on-demand migrations.

IV. Evaluation

Although performance is not the main objective of SVM, it is important to examine how our SVM framework performs in comparison to a conventional FPGA execution framework. In the following, we compare the runtime of our SVM framework when using page fault triggered On-Demand Page Migrations (ODPMs), User Managed Page Migrations (UMPMs) in *both* directions and UMPMs for back-migrations to host memory *only*, to the runtime when using the existing TaPaSCo framework [2]. TaPaSCo uses conventional DMA copy-based memory management, and operates with physical addresses. We use a VC709 evaluation board [38] connected via PCIe 3.0 with eight lanes and $16\,\mathrm{Gbit/s}$ bandwidth to an AMD Athlon X4 845 Quad Core with $16\,\mathrm{GB}$ host memory. The Level 1 TLB of our on-FPGA IOMMU is in 32-entry configuration, while the Level 2 TLB provides 1024 entries, which are split in 32 sets of 32 ways each, and spread over four BRAMs.

Our evaluated benchmarks represent different common memory access scenarios. However, the performance measurement taken here does not reflect the much reduced programming complexity due to the use of SVM. Especially applications with pointer-rich data structures benefit from SVM, as it potentially reduces code size, complexity, and error-proneness. Hence, even a runtime similar to using conventional memory management schemes is generally already a success when taking the other advantages of SVM into account.

A. ArraySum

The *ArraySum* benchmark calculates the sum over $64\,\mathrm{bit}$ values in an array, and represents scenarios where an accelerator iterates once over an array with high data throughput. Figure 3 shows the measured runtimes normalized to the runtime when using TaPaSCo for different array sizes. All runtimes are the mean of 10,000 measurements.

It is not surprising that SVM cannot reach the runtimes of TaPaSCo, since in this scenario, most of the time is spent for migrating the data to device memory. SVM with page migrations introduces additional overhead during the migration process for a number of reasons:

- A PTW is performed to collect the pages to be migrated.
- Page tables and TLBs in the system must be updated.
- The pages are often distributed in memory at host side, and sometimes at device side as well when using ODPMs. This requires scatter-gather DMA transfers and decreases the PCIe throughput.

For small buffer sizes, ODPMs perform worst. In the case of a $4\,\mathrm{kB}$ buffer, which we assume is not page aligned and thus crosses page boundaries, this requires two pages to be migrated due to the misalignment of the buffer. Here, ODPMs take more than 2.5 times the runtime of TaPaSCo. ODPMs are also significantly slower than UMPMs, although UMPMs cannot reduce overhead much in this case, since the two page faults are handled together by the same worker thread as well. However, it takes some time until the IOMMU interrupt arrives after launching the accelerator ($\sim 2.5\,\mu\mathrm{s}$), and the worker thread is eventually scheduled ($\sim 5.5\,\mu\mathrm{s}$). Also, the fault handling thread performs an additional PTW to check whether the page is already located in device memory. With increasing buffer size, the performance of ODPMs improves, and reaches an only $25\,\%$ higher runtime than TaPaSCo for $32\,\mathrm{MB}$ buffers. In contrast, the performance of UMPMs has its maximum for a $32\,\mathrm{kB}$ buffer with only 1.1 times the runtime of TaPaSCo. UMPMs can reduce overhead compared to ODPMs

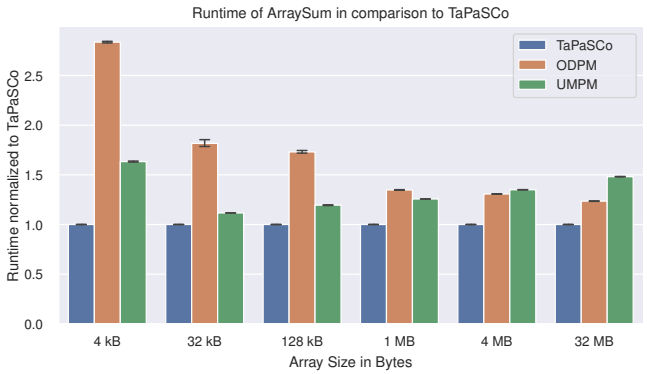


Fig. 3. Runtime of ArraySum with TaPaSCo and SVM. All results are normalized to the mean runtime when using just TaPaSCo without any SVM support.

by migrating all pages of a buffer together. However, with increasing buffer size, the less efficient scatter-gather DMA transfer has more impact on the runtime in comparison to TaPaSCo. On the other hand, ODPMs have the advantage over UMPMs and TaPaSCo that the accelerator can start its computation immediately after the first page has been migrated, instead of waiting to be launched only after all data is located in device memory. Hence, ODPMs achieve lower runtimes than UMPMs for larger buffer sizes in this scenario.

B. MemCopy

In the *MemCopy* scenario, the accelerator simply copies data from one buffer to another. In order to get more insights, we vary not only the array size, but also the number of copy iterations. This simulates scenarios in which an accelerator has to iterate over its input data more than once. Again, the maximum data throughput is limited by the memory bandwidth rather than the accelerator's throughput.

The normalized runtimes of *MemCopy* are shown for three different array sizes in Figure 4. Due to long total runtimes, we use the mean of 100 runs, which leads to slightly larger 95 % confidence intervals. The unusually wide confidence interval for the case of 32 kB buffer size and four copy iterations is due to a single of the 100 executions having a runtime 20 times longer than usual. The reasons for this outlier are still unknown to us.

A direct comparison to the *ArraySum* scenario shows longer runtimes for all SVM variants in the *MemCopy* scenario. TaPaSCo only copies the input buffer to device memory, and the output buffer back to host memory. When using SVM, the output buffer is allocated with `malloc()` in host memory, and must be migrated to device memory as well, since the framework cannot know that the buffer is uninitialized, and will in the end only be written to. As *MemCopy* also requires a back-migration, we differentiate in the graphs between runtimes with UMPMs in both directions, and back to host memory only.

The overall picture is similar for all array sizes in this scenario. For a low number of copy iterations, TaPaSCo outperforms SVM. Using UMPMs for at least back-migrations already shortens runtimes significantly. When increasing the number of copy iterations, however, the runtimes of all SVM variants converge to the runtime of TaPaSCo for the array sizes of 32 kB and 1 MB. With 1 MB array size, 128 copy iterations and UMPMs, we even achieve an about 15 % faster runtime. The reason is the ratio between accelerator-only and total runtime, which is shown for the TaPaSCo case in Table II. When using few copy iterations, the data migration time clearly dominates the runtime, but in the case of 128 iterations, the accelerator computes for more than 90 % of the total runtime. The impact of the additional overhead for page migrations is negligible then. In fact, our framework achieves an even higher maximum memory bandwidth than TaPaSCo. Hence, the additional address translations in our device IOMMU do not affect the maximum achievable bandwidth at all.

TABLE II
RATIO BETWEEN ACCELERATOR-ONLY RUNTIME AND TOTAL RUNTIME IN THE MEMCOPY SCENARIO WHEN USING TAPASCO.

Array size	Iterations	Accelerator-only	Total runtime	Ratio
32 kB	1	0.013 ms	0.103 ms	12.6 %
32 kB	128	1.625 ms	1.765 ms	92.1 %
1 MB	1	0.406 ms	2.259 ms	17.9 %
1 MB	128	51.990 ms	53.780 ms	96.7 %
32 MB	1	12.997 ms	59.086 ms	22.0 %
32 MB	128	1663.670 ms	1709.910 ms	97.3 %

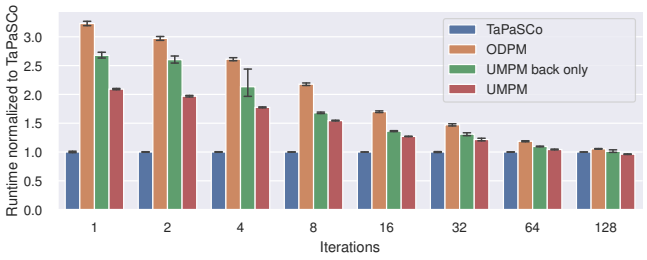
The results for 32 MB array size differ in one detail. Here, the runtimes of both SVM variants using ODPMs in host-to-device direction settle at around 1.7 times the TaPaSCo runtime for large numbers of copy iterations. In this case, the capacity of the Level 2 TLB is not sufficient to hold mappings for *all* memory pages in this scenario. Hence, the TLB entries get evicted, and the device IOMMU must issue new page faults in every copy iteration to ask the driver for address translations and re-adding the TLB mappings. However, when using UMPMs for *host-to-device* transfers as well, we are able to use the specialized TLB for arbitrary length mappings (see Section III-B3), which has a higher capacity, and achieve similar runtimes as with TaPaSCo.

The *MemCopy* scenario shows that with increasing data reuse overhead due to SVM decreases, and SVM can compete with conventional memory management schemes.

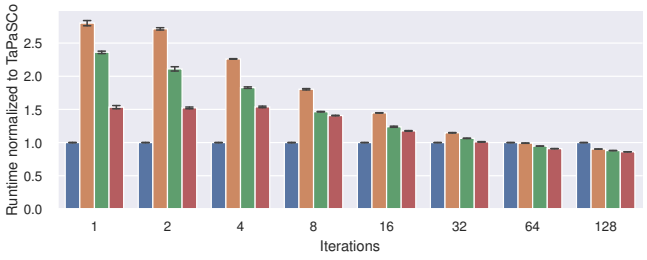
C. Sobel Filter

The *Sobel Filter* is an image processing filter used for edge detection algorithms. In contrast to the previous benchmarks, the data throughput of the accelerator is the limiting factor for the overall throughput in this case. In our evaluation, we compare an IP core generated with Vitis HLS [42] to a custom core written in Bluespec SystemVerilog (BSV). The HLS core has a fixed image size of 512x512 px, while our custom accelerator works on different image sizes from 512x512 to 1920x1080 px.

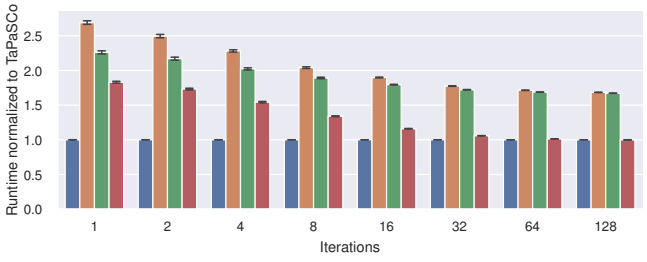
The runtimes with TaPaSCo and the different SVM variants are shown in absolute numbers in Figure 5. When looking at



(a) Array size = 32 kB



(b) Array size = 1 MB



(c) Array size = 32 MB

Fig. 4. Runtime of MemCopy with different array sizes and numbers of copy iterations. The results using SVM are normalized to the mean runtime when using just TaPaSCo without SVM support.

the BSV core, ODPMs perform better than UMPMs for all image sizes, in contrast to the *MemCopy* scenario. Due to the moderate data throughput of 200 MB/s, the framework has sufficient time to migrate pages in the background during the computation, and ODPMs benefit from the fact that the computation can start immediately after the first page is present in device memory. When using UMPMs, all pages are migrated before launching the accelerator, and we encounter the same slowdown compared to TaPaSCo as we have seen in the previous benchmarks. If we combine ODPMs in host-to-device migration with UMPMs in device-to-host direction after the computation, SVM achieves almost the same runtime as TaPaSCo, despite the additional migration overhead.

The results for the HLS core show another picture. The TaPaSCo and UMPMs runtimes are similar to the BSV core. However, the other two SVM variants need more than *double* the runtime, and the explicit back-migration does not give a significant performance boost. This discrepancy is caused by the different memory access patterns of both cores. Our

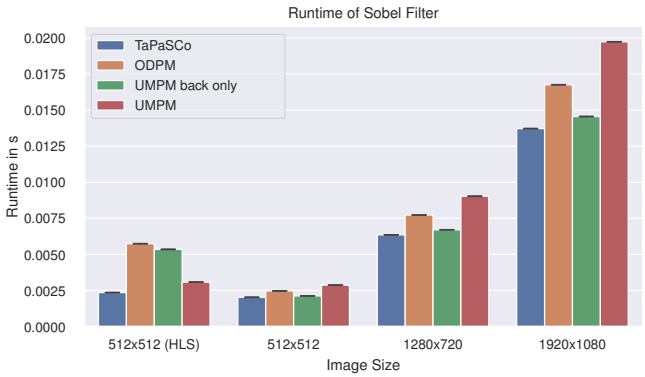


Fig. 5. Runtime of the Sobel Filter HLS and BSV cores with TaPaSCo and SVM.

custom BSV core utilizes the maximum AXI4 burst length [39], and covers an entire 4 kB memory page with a single read or write request. Also, it issues requests to memory as early as possible. Hence, the device IOMMU is able to prefetch up to 16 pages in advance, and the SVM framework has sufficient time to migrate pages in the background until the accelerator actually requires the respective data. In contrast, the HLS core uses shorter bursts with 64 B of length, and only issues a few requests in advance. This has limited impact if the data is already located in device memory, but in the SVM case the device IOMMU is only able to fault-in one page in advance. In addition, the HLS core requires the data shortly after requesting it, and thus has to stall and wait until the requested page is migrated.

The *Sobel Filter* scenario shows that in scenarios with moderate data throughput, SVM achieves similar runtimes to conventional memory management schemes, since the framework has sufficient time to migrate data while the accelerator is already running. However, it is important that the accelerator issues memory requests sufficient far in advance to give the SVM framework a chance to react. Additionally, we see in the *Sobel Filter* results that UMPMs in device-to-host direction are favorable in most cases, since the chance to benefit from the one-by-one back-migration performed on-demand is very low.

D. Overhead Evaluation

In order to evaluate where the already described overhead occurs, we split the migration into four steps:

- *Setup:* The HMM API collects the pages and invalidates the system TLBs, the driver invalidates the IOMMU TLB, and checks for ongoing memory accesses (device-to-host migration only)
- *Allocate:* The driver allocates pages in destination memory and prepares the DMA transfer
- *DMA:* The DMA engine copies the data from source to destination memory
- *Finalize:* The driver frees the pages in source memory, the HMM API updates the page table, the driver adds IOMMU TLB entries (host-to-device migration only)

TABLE III
RUNTIMES OF THE DIFFERENT MIGRATION STEPS IN HOST-TO-DEVICE AND DEVICE-TO-HOST DIRECTION FOR DIFFERENT BUFFER SIZES.

Buffer	32 kB		1 MB		32 MB	
Host-to-device buffer migration						
Setup	6.52 µs	11.4 %	0.10 ms	8.8 %	2.12 ms	6.3 %
Allocate	3.43 µs	6.0 %	0.07 ms	6.0 %	2.06 ms	6.2 %
DMA	38.14 µs	66.4 %	0.76 ms	68.4 %	23.32 ms	70.0 %
Finalize	9.33 µs	16.2 %	0.19 ms	16.7 %	5.83 ms	17.5 %
Device-to-host buffer migration						
Setup	18.75 µs	27.8 %	0.40 ms	30.3 %	12.55 ms	31.2 %
Allocate	4.49 µs	6.6 %	0.09 ms	6.6 %	2.68 ms	6.7 %
DMA	34.29 µs	50.7 %	0.65 ms	49.1 %	19.63 ms	48.7 %
Finalize	10.03 µs	14.8 %	0.18 ms	14.0 %	5.41 ms	13.4 %

Table III states the times of the different migration steps for different buffer sizes. In host-to-device direction, the DMA transfer is clearly the most time consuming step with a fraction of up to 70 %. While the efforts for *Allocate* and *Finalize* do not vary significantly with a changing buffer size, the part of the *Setup* phase decreases with increasing buffer size. A reason may be that the PTW for collecting the pages to be migrated becomes more efficient when iterating over more page entries.

In device-to-host direction, the DMA transfer still requires the longest time. However, it takes only about *half* of the total migration time. In contrast to host-to-device migrations, the fraction of the *Setup* phase is significantly higher and ranges between 27 % and 31 %. The driver has two additional tasks during back-migrations: (1) invalidating the respective TLB entries, and (2) checking or even waiting for active memory requests to the desired page. Currently, the driver checks for memory accesses to every page separately. However, this could be optimized in future versions to save some time, especially when migrating larger buffers.

The DMA transfer takes not only the largest fraction, but also needs more absolute time compared to conventional memory management schemes. A conventional driver usually places all data in a single large buffer, and the DMA engine can copy it in one go to device memory or back. In our SVM case however, the source and destination pages may be scattered across different memory regions. Hence, multiple small transfers are often required, and decrease the overall efficiency. However, we achieve more than the double performance with our custom *PageDMA* IP core in comparison to Xilinx's XDMA [43] engine during ODPMs.

V. CONCLUSION AND FUTURE WORK

Our SVM framework with physical page migrations can considerably simplify the use of hardware accelerators from software programs. The programmer may not only pass virtual addresses directly to the accelerator, but the page fault capabilities of our device IOMMU also alleviate the need for explicitly moving the data. In contrast to related work, our framework targets standard PCIe-attached FPGA cards while using their on-device memory, and does not require dedicated hardware support on host side.

However, SVM carries an overhead during data migration to and from device memory. The driver has to perform a PTW to collect the pages to be migrated, and update page table entries to indicate their new physical location. Furthermore, source and destination pages may be scattered across different memory regions, which leads to many small DMA transfers. In host-to-device direction, the driver reduces the migration overhead by coalescing up to 16 page faults. However, device-to-host migrations are triggered by the OS kernel for single pages, and must be performed page by page. The feature of UMPMs helps in many scenarios. Here, the programmer explicitly manages the migration of an entire buffer to or from device memory.

Our evaluation in Section IV shows that scenarios with high data throughput, but limited data re-use are unfavorable for SVM. The runtimes of these scenarios are dominated by the time required to migrate the data, which is the weakness of SVM due to the described, additional overhead. However, if data re-use becomes more pronounced, the impact of the migration overhead shrinks, and SVM achieves comparable runtimes to conventional memory management schemes. In some cases, we even achieve a higher maximum bandwidth than the reference framework TaPaSCo [2], which clearly shows that the address translations in our device IOMMU do not affect the memory bandwidth itself. Applications with moderate data throughput benefit from the fact that the accelerator can start its computation immediately after the first page has been migrated. The following memory pages can then be migrated in the background while the accelerator is already running. Here, it is beneficial if the accelerator issues memory requests far in advance of the data uses to give the SVM framework sufficient time to migrate the requested data.

Much of overhead is caused by the default small 4 kB page size. The use of 2 MB huge pages would reduce the overall number of page faults, and increase the efficiency during migrations significantly. Additionally, the TLB capacity would be much higher, since every entry would map 2 MB instead of 4 kB. However, the HMM API [25] does not support huge pages currently. Implementing a workaround for huge pages *without* the HMM API would not be practical, since that would create two parallel solutions in one framework, leading to much duplicated implementation effort.

The evaluation also shows that TLB evictions when handling large data structures are costly, since currently the device driver has to be asked for translating the virtual addresses with help of the CPU page tables. A way for the device IOMMU to perform address translations on its own would relieve the pressure on the TLBs. One solution would be the use of the Address Translation Service of PCIe. An even faster solution could be a private page table in device memory. In this manner, the IOMMU could perform PTWs autonomously, such as in [5], [7], [13], and TLB evictions would be significantly less costly.

We will integrate this work as an additional feature into the open-source framework TaPaSCo, and publish all source code in the Github repository [44].

References

[1] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, p. 48–60, Jan. 2019.

[2] C. Heinz, J. Hofmann, J. Korinth, L. Sommer, and A. Koch, "The TaPaSCo open-source toolflow," *J Sign Process Syst*, no. 93, pp. 545–563, 2021.

[3] Xilinx Inc., "Zynq-7000 SoC data sheet: Overview." [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. [Accessed: 20-Apr-2021].

[4] Intel Corp., "Cyclone V device datasheet." [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_51002.pdf. [Accessed: 20-Apr-2021].

[5] H. Lange and A. Koch, "Architectures and execution models for hardware/software compilation and their system-level realization," *IEEE Transactions on Computers*, vol. 59, no. 10, pp. 1363–1377, 2010.

[6] Xilinx Inc., "ML310 user guide." [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/ug068.pdf. [Accessed: 26-Jul-2021].

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

[8] Intel Corp., "OpenCL 2.0 shared virtual memory overview." [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/opencl-20-shared-virtual-memory-overview.html. [Accessed: 21-Apr-2021].

[9] 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, 2019.

[10] B. Klauer, "The convey hybrid-core architecture," *High Performace Computing Using FPGAs*, pp. 431–451, 2013.

[11] Convey Computer Corporation, "Convey programmers guide," Nov. 2010.

[12] 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)*, pp. 458–461, 2013.

[13] 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, 2015.

[14] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta, "A reconfigurable computing system based on a cache-coherent fabric," *2011 International Conference on Reconfigurable Computing and FPGAs*, pp. 80–85, 2011.

[15] G. Alonso, T. Roscoe, D. Cock, M. Ewaida, K. Kara, D. Korolija, D. Sidler, and Z. Wang, "Tackling hardware/software co-design from a database perspective," *10th Annual Conference on Innovative Data Systems Research (CIDR '20)*, Jan. 2020.

[16] A. Ramdas, D. Cock, T. Roscoe, and G. Alonso, "The Enzian Coherent Interconnect (ECI): opening a coherence protocol to research and applications," *LATTE '21*, Apr. 2021.

[17] T. P. Morgan, "A peek inside that Intel Xeon-FPGA hybrid chip." [Online]. Available: https://www.nextplatform.com/2018/05/24/a-peek-inside-that-intel-xeon-fpga-hybrid-chip/. [Accessed: 19-Jul-2021].

[18] Y.-K. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "In-depth analysis on microarchitectures of modern heterogeneous CPU-FPGA platforms," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, Feb. 2019.

[19] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, and B. Allison, "IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI," *IBM Journal of Research and Development*, vol. 62, no. 4/5, pp. 8:1–8:8, 2018.

[20] CXL Consortium, "Compute express link." [Online]. Available: https://www.computeexpresslink.org. [Accessed: 20-Apr-2021].

[21] CCIX Consortium, "Cache coherent interconnect for accelerators." [Online]. Available: https://www.ccixconsortium.com. [Accessed: 20-Apr-2021].

[22] M. Harris, "Unified memory in CUDA 6." [Online]. Available: https://developer.nvidia.com/blog/unified-memory-in-cuda-6/. [Accessed: 21-Apr-2021].

[23] HSA Foundation, "HSA platform system architecture specification, version 1.0." [Online]. Available: http://www.hsafoundation.com/?ddownload=4944, Jan. 2015. [Accessed: 19-Jul-2021].

[24] NVIDIA Corp., "Data center drivers for linux x64 download." [Online]. Available: https://www.nvidia.com/Download/driverResults.aspx/173142/en-us. [Accessed: 19-Jul-2021].

[25] The Linux Kernel Developement Community, "Heterogenous Memory Managemet API." [Online]. Available: https://www.kernel.org/doc/html/latest/vm/hmm.html. [Accessed: 19-Apr-2021].

[26] The Linux Kernel Developement Community, "Nouveau GPU driver source code." [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/gpu/drm/nouveau?h=v5.10.33. [Accessed: 29-Apr-2021].

[27] The Linux Kernel Developement Community, "AMD GPU driver source code." [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/gpu/drm/amd?h=v5.10.33. [Accessed: 19-Jul-2021].

[28] M. Larabel, "Intel revises the shared virtual memory support for their linux graphics driver." [Online]. Available: https://www.phoronix.com/scan.php?page=news_item&px=Intel-SVM-V2-For-Linux. [Accessed: 19-Jul-2021].

[29] A. K. Ziabari, Y. Sun, Y. Ma, D. Schaa, J. L. Abellán, R. Ubal, J. Kim, A. Joshi, and D. Kaeli, "UMH: A hardware-based unified memory hierarchy for systems with multiple discrete GPUs," *ACM Trans. Archit. Code Optim.*, vol. 13, Dec. 2016.

[30] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining HW/SW mechanisms to improve NUMA performance of multi-GPU systems," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 339–351, 2018.

[31] T. Baruah, Y. Sun, A. T. Dinçer, S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli, "Griffin: Hardware-software support for efficient page migration in multi-GPU systems," *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 596–609, 2020.

[32] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: A GPU memory manager with application-transparent support for multiple page sizes," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), p. 136–150, Association for Computing Machinery, 2017.

[33] S. Zhang, Z. Qin, and Y. e. a. Yang, "Transparent partial page migration between CPU and GPU," . *Front. Comput. Sci.*, no. 14, 2020.

[34] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Adaptive page migration for irregular data-intensive applications under GPU memory oversubscription," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 451–461, 2020.

[35] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Interplay between hardware prefetcher and page eviction policy in CPU-GPU unified virtual memory," in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), p. 224–235, Association for Computing Machinery, 2019.

[36] D. Korolija, T. Roscoe, and G. Alonso, "Do OS abstractions make sense on FPGAs?," *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 991–1010, Nov. 2020.

[37] D. Korolija, "Coyote driver source code on Github." [Online]. Available: https://github.com/fpgasystems/Coyote/tree/master/driver. [Accessed: 19-Jul-2021].

[38] Xilinx Inc., "VC709 evaluation board for the Virtex-7 FPGA user guide." [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/vc709/ug887-vc709-eval-board-v7-fpga.pdf. [Accessed: 20-Apr-2021].

[39] ARM Ltd., "AMBA AXI and ACE protocol specification." [Online]. Available: https://documentation-service.arm.com/static/602a9df190ee6824a1e02b98. [Accessed: 20-Apr-2021.

[40] The Linux Kernel Developement Community, "HMM test module." [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/gpu/drm/nouveau?h=v5.10.33. [Accessed: 29-Apr-2021].

[41] T. Heo and F. Mickler, "Concurrency managed workqueue." [Online]. Available: https://www.kernel.org/doc/html/latest/core-api/workqueue.html?highlight=cmwq. [Accessed: 30-Apr-2021].

[42] Xilinx Inc., "Vitis high-level synthesis." [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. [Accessed: 05-May-2021].

[43] Xilinx Inc., "DMA/Bridge subsystem for PCI express v4.1 product guide." [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/xdma/v4_1/pg195-pcie-dma.pdf. Accessed: 26-Apr-2021.
[44] Embedded Systems and Applications Group, TU Darmstadt, "Tapasco on Github." [Online]. Available: https://github.com/esa-tu-darmstadt/tapasco. [Accessed: 19-Jul-2021].