ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Vogel, P., Marongiu, A., Benini, L. (2019). Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU. IEEE TRANSACTIONS ON COMPUTERS, 68(4), 510-525 [10.1109/TC.2018.2879080].

Availability:
This version is available at: https://hdl.handle.net/11585/651834 since: 2018-12-06

Published:

DOI: http://doi.org/10.1109/TC.2018.2879080

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

16 November 2026

This is the post peer-review accepted manuscript of:

P. Vogel, A. Marongiu, L. Benini, “Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU” 2018 IEEE Transactions on Computers (TC), Early access, DOI: 10.1109/TC.2018.2879080

The published version is available online at: https://doi.org/10.1109/TC.2018.2879080

© 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works

Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU

Pirmin Vogel, *Student Member, IEEE,* Andrea Marongiu, *Member, IEEE,* and Luca Benini, *Fellow, IEEE*

Abstract—A key enabler for the ever-increasing adoption of FPGA accelerators is the availability of frameworks allowing for the seamless coupling to general-purpose host processors. Embedded FPGA+CPU systems still heavily rely on copy-based host-to-accelerator communication, which complicates application development.
In this paper, we present a hardware/software framework for enabling transparent, shared virtual memory for FPGA accelerators in embedded SoCs. It can use a hard-macro IOMMU if available, or a configurable soft-core IOMMU that we provide. We explore different TLB configurations and provide a comparison with other designs for shared virtual memory to gain insight on performance-critical IOMMU components. Experimental results using pointer-rich benchmarks show that our framework not only simplifies FPGA-accelerated application development, it also achieves up to 13x speedup compared to traditional copy-based offloading.

Index Terms—Shared Virtual Memory, FPGA Accelerators, Heterogeneous SoCs, Embedded Systems.

1 Introduction

Over the past decade, field-programmable gate arrays (FPGAs) have gained importance as a result of the big shift towards cloud computing, where energy efficiency, scalability and throughput are critical [1] [2].

Besides substantially increased performance and logic/memory density, several advancements targeting usability are at the basis of this success. Modern FPGA design environments unify the relevant tools in a single framework and provide a large set of intellectual property (IP) cores allowing FPGA designers to focus on the more critical core modules (e.g., custom hardware engines).

In the high-performance computing (HPC) domain sophisticated third-party frameworks such as IBM CAPI [3], Microsoft Project Catapult [4] and Convey's Hybrid-Core Architecture [5] allow to seamlessly integrate FPGA operation into the execution model of general-purpose host processors. By granting the FPGA coherent access to the virtual memory of the host, these frameworks enable "holistic" approaches to HW/SW partitioning, where the FPGA accelerates critical parts of applications started on the host, explicitly addressing the data from the same virtual memory space. This tremendously simplifies FPGA-accelerated application development and deployment, and usually also improves performance, as the need for continuous data movements between separate address spaces is lifted [3].

In the embedded systems world, the situation is different [6] [7]. While for quite some time the major FPGA vendors have had devices on the market that combine multi-core, general-purpose host processors with FPGA fabrics in heterogeneous systems-on-chip (SoCs) [8] [9], shared virtual memory (SVM) is still not widely adopted.

For embedded systems that allow the FPGA accelerator itself [10] [11] to proactively fetch data from main into local FPGA memory, the state of the art is still copy-based shared memory. The main memory is statically split into two sections: one exclusively accessed by the host via cached, paged virtual addressing, and a second that is accessed by both the host and the FPGA via uncached, contiguous physical addressing. Before and after every computation offloading to the FPGA, the host must copy the relevant data between the two memory sections. This approach requires minimal hardware support, but comes at the expense of several substantial drawbacks. First, besides the cost for the continuous data copies, the static splitting of the main memory is far from optimal, particularly considering the limited memory capacity of embedded systems and the constraints it poses on the size of the shareable dataset. Second, copy-based data sharing is not suitable for the acceleration of pointer-rich applications, as any virtual-address pointers inside the shared data need to be modified to point to the copy in the physically contiguous section, requiring a complete rewrite of the accelerated code and application-specific offload sequences. Third, the performance of the host code is also reduced, as to avoid coherency problems mutually exclusive access to the shared data is usually employed.

Recently, high-end FPGA-based heterogeneous SoCs like the Xilinx Zynq UltraScale+ MPSoC, the *ZynqUS+*, have started appearing on the marketplace – in the form of engineering samples [12]. These sophisticated systems feature hardware input/output memory management units (IOMMU) [13], which could simplify the adoption of SVM for FPGA accelerators. However, unlike the frameworks adopted in the HPC domain [3] [4] [5], there is no infrastructure available to help application programmers bridging the gap between their user-space applications and the Linux kernel for leveraging the IOMMU hardware for SVM. The available software support is limited to just a

- *P. Vogel and L. Benini are with the Integrated Systems Laboratory at ETH Zurich, Switzerland, {vogel, benini}@iis.ee.ethz.ch*
- *A. Marongiu is with the Department of Computer Science and Engineering of the University of Bologna, Italy, a.marongiu@unibo.it*
- *L. Benini is with the Electrical, Electronic, and Information Engineering Department of the University of Bologna, Italy, luca.benini@unibo.it*

Manuscript received February 5, 2018; revised August 5, 2018.

low-level hardware driver exposing a limited set of features to the kernel. The main usage scenario for the IOMMU is currently the protection of the host from malicious or faulty DMA devices and drivers. This is evident from the functionality exposed by the platform development tools [14], which only contemplate IOMMU operation in combination with host-initiated DMA transfers. To enable coherent and direct FPGA access to virtual user-space memory a custom device driver is needed to interface the actual hardware driver through the Linux IOMMU API, plus extensions to the API itself to allow the selection of a specific process' page table. To hide the low-level details of the internals of such a driver, more abstract programming interfaces like an offloading runtime environment must be provided.

In this work, we present a plug-and-play framework for exploring SVM support for FPGA accelerators in embedded SoCs. It consists of two core components shown in Fig. 1 a):

1) A software SVM manager running on the host, consisting of a kernel-level driver module that uses standard Linux kernel APIs to manage the IOMMU hardware, and a user-space runtime library interfacing this driver with the upper software layers (e.g., programming models' runtime libraries) and ultimately to the application to be accelerated;
2) a parameterized and configurable IOMMU design that can be deployed on the FPGA and interfaced to the accelerator to enable SVM communication. This soft-core IOMMU enables SVM on SoCs that lack an equivalent hard-macro IOMMU, but can also enable interesting performance boosts if a hard IOMMU is available as it supports mappings of flexible size and non-blocking behavior (see Section 5.3).

This framework enables transparent SVM for FPGA accelerators. Sharing data becomes as simple as passing a virtual address pointer to the FPGA accelerator. The accelerator itself can fetch the data from SVM, which allows it to directly operate on pointer-rich data structures without relying on the host for data management.

Concerning the soft-core IOMMU, we explore different design choices for the most critical IOMMU building block: the translation lookaside buffer (TLB). First, we consider a fully-parallel TLB with optimized look-up latency (1 cycle), in accordance to all state-of-the-art designs. A fully-associative design enables variable sized mappings for flexible operation. Similar to any fast memory, the downside of such a solution is poor scalability to large sizes, which ultimately translates in higher page miss rates experienced by the FPGA accelerator. Motivated by the observation that in many cases accelerator traffic is more bandwidth-sensitive than latency-sensitive, we consider a second design that tolerates larger look-up times to enable much larger TLB sizes (lower miss rate). We call these two designs the level-1 (L1) and level-2 (L2) TLB, respectively, following the traditional latency/size-based classification in memory hierarchies. We also explore a third variant, which we call the "hybrid" TLB and which aims at combining the best of both schemes.

We evaluate the performance of our framework both when using our own IOMMU or the hard-macro IOMMU in ZynqUS+. To this end, we interface it to an accelerator generating the memory access patterns of real-world applications, including memory-bound, compute-bound, and pointer-chasing kernels. To broaden the scope of the exploration we have parameterized such kernels to represent instances operating on varied input datasets. The evaluation provides deep insight on SVM design for FPGA accelerators.

1) Compared to copy-based memory sharing, our SVM framework allows for speedups between 1.5 and 13x for purely memory-bound kernels.
2) Due to limitations in the low-level drivers and kernel APIs, the performance of hard-macro IOMMUs can be dominated by page fault handling.
3) Thanks to its higher flexibility, our soft IOMMU can achieve better performance at negligible FPGA resource cost even if a hard IOMMU is available.
4) Non-blocking queuing of multiple outstanding misses and batch-mode handling is beneficial for parallel accelerator architectures and can improve performance over designs using costly, dedicated hardware for low-latency miss handling.
5) TLB look-up latency is not critical for FPGA accelerators as the TLB is not in the critical path of the accelerator to the internal scratchpad memories (SPMs). Instead, address translation is primarily needed for the latency-insensitive DMA transfers used to copy data between these SPMs and SVM. Relaxing look-up latency allows for the construction of larger TLBs and thus lower miss rate (and overall miss-handling overhead) with less FPGA resources.

The remainder of this paper is as follows. Section 2 discusses related work. The implementation of our design is presented in Section 3. Section 4 gives the details of our evaluation platforms. In Section 5, we discuss the results of our evaluations. Section 6 concludes the paper.

2 Related Work

Shared virtual memory (SVM) has been widely studied in the context of heterogeneous embedded SoCs based on programmable many-core accelerators (PMCAs) such as graphics processing units, where it is being increasingly adopted [15], [16]. In this field, the research community has evaluated commercial designs [17] and also proposed optimized SVM infrastructure for custom PMCA architectures [18], [19]. In our previous works, we have explored lightweight SVM support for PMCAs based on a software-managed IOTLB considering applications based on regular [20] and irregular (pointer-rich) [21] memory access patterns, and exploring PMCA-local IOTLB management [22].

With this work, we shift our focus on a different type of accelerator, namely generic, custom hardware accelerators deployed on FPGA. Where in our previous works we have used FPGA logic as a substrate to implement an emulator of a PMCA-based heterogeneous SoC (by deploying RTL models), in this work we target the FPGA as a medium for true computation acceleration. While unavoidably there are some commonalities between this prior work and the software stack as well as the architecture of the L1 TLB that we propose in this paper, focusing on FPGA accelerators requires brand new design solutions and evaluations. The proposed software stack enables SVM both when using the

hard-macro IOMMU available on some next-gen high-end FPGA-based SoCs or our own configurable soft IOMMU. For this soft IOMMU, we propose a hybrid TLB design combining a flexible L1 TLB with a new, scalable and configurable L2 TLB optimized for FPGA deployment, and that supports TLB prefetching transactions. This design allows to increase TLB capacity by 16x and more compared to related works, while achieving lower overall resource cost and higher or comparable clock frequencies. We provide an in-depth analysis (previously never covered) of i) the effect of TLB look-up latency and capacity on performance; ii) the trade-off for cache-coherent accelerator accesses to SVM; iii) a comparison with hard-macro IOMMUs and alternative SVM designs from literature that demonstrates the benefits of our hybrid design, the prefetching transactions and non-blocking TLB misses for parallel FPGA accelerators.

In the HPC domain, SVM for letting FPGA accelerators themselves orchestrate data transfers from and to main memory has proven beneficial both for performance and programmability [3] [23]. While academic works focus on SVM solutions consisting of TLBs managed in software either running on the host [23] or on a dedicated soft processor core [24], the industry's approach is that of full-blown hardware for maximum performance [3] [4] [5]. To optimize off-chip bandwidth utilization between host and FPGA, such systems employ FPGA-side data caches, as well as transaction coalescing and reordering [25], which further increases design complexity and leads to considerable resource utilization. For example, IBM's CAPI utilizes around 25% of the resources provided by a medium-sized, high-end FPGA for data centers [26]. The same design would almost completely occupy even the largest FPGAs found in embedded SoCs, leaving little resources for the actual accelerator only. Clearly, the same level of hardware support is not feasible for embedded systems.

Some HPC systems also use private accelerator DRAM to avoid the latency and bandwidth bottleneck between host and FPGA, and they rely on special APIs to allocate pinned host memory for maximum accelerator performance. This complicates application development as pinned memory leads to higher management overheads for the host [25]. In contrast, FPGA fabrics in embedded SoCs are tighter integrated to the memory system of the host, leading to lower latency and higher bandwidth to main memory. This diminishes the benefits of additional, private DRAM but moves the SVM interface more in the focus for accelerator performance as it is used for all accesses to off-chip memory.

Current-generation FPGA-enabled, embedded SoCs [8] [9] do not provide SVM support in the form of IP cores and associated driver software. To enable SVM for FPGA accelerators designed with HLS, a recent study [27] proposes an HLS framework extension to provide each shared data element with its private address translation hardware. At offload time, a kernel-level driver on the host locks all memory pages touched by these data elements and creates an optimized translation table for the hardware. While the design allows to tailor the SVM subsystem to the application at hand, it provides only little run-time flexibility and it is not usable when operating on pointer-rich data structures.

Another approach relies on operating system support to enable SVM using per-thread hardware IOMMUs [28]. The focus of this large and versatile framework primarily lies on providing the application developer a uniform view of software threads executing on the host and hardware threads mapped to FPGA logic. The SVM subsystem is however just a little component inside a large framework. As a consequence, compared to our proposal i) its internals have not been studied in depth and cannot simply be decoupled from the rest of the framework to be used with custom FPGA accelerators; ii) the interaction between hard- and soft-threads is not as streamlined as ours, being it designed on top of less lightweight interfaces such as POSIX threads.

Most designs stick to more basic accelerator interaction and memory sharing models [6] [7], where the shared data is placed in contiguous memory using a specific user-space API [18] or by replacing the standard *malloc()* with a customized implementation [19]. Address translation is performed explicitly by the host as part of the DMA transfer preparation from contiguous main memory to the accelerator's local memories [14] [29] [30]. This model where the host is responsible for the explicit management of the accelerator memory is not suitable for applications that operate on pointer-rich data structures or perform fine to medium-grained offloads [29]. Moreover, contiguous memory allocation has several other drawbacks, as the Linux implementation has shown [31]. For example, if the kernel must first copy other data out of the pre-allocated, contiguous memory region before it can be used for shared data, a long latency results. Moreover, there is no guarantee that the pre-allocated region can be freed and made available at run time. Finally, CMA returns uncached memory: letting the host operate on such memory is very inefficient.

Ideally, a full-fledged hardware IOMMU [13] [32] – similar to what is nowadays found in high-end SoCs based on PMCAs [15] [16] – provides the required functionality to enable true SVM for custom FPGA accelerators in the embedded systems domain. Opting for maximum performance and complete abstraction of the underlying SVM system, these designs include hardware page-table walkers, coherent translation caches and large buffers to absorb memory transactions including DMA transfers missing in the TLB.

While with the ZynqUS+, the next-generation SoC featuring a hard-macro IOMMU is becoming available [12], embedded SoCs lack in infrastructure software that lets programmers interface their user-space applications with the kernel's IOMMU API and low-level hardware drivers. In addition, modifications to the ARM-specific implementation of this kernel API are required to let the IOMMU directly operate on the process page table instead of creating an empty I/O page table upon IOMMU initialization, similar to some desktop-class systems with IOMMU support.[1] Without the latter, the handling of page faults in this I/O page table, which has to be carried out by software running on the host, can quickly become the major bottleneck. Currently, it is not foreseen in ZynqUS+ to use the IOMMU for giving

1. ARM has released experimental kernel patches that aim at 1) supporting SVM in ARM-based systems and 2) unifying different AMD-, Intel- and ARM-specific IOMMU API extensions for SVM. However, these patches have not yet been merged into mainline Linux and more importantly, they only target future ARM IOMMU architecture revisions and are not compatible with current-gen IOMMU devices as the one in ZynqUS+. See: git://linux-arm.org/linux-jpb.git svm/rfc1

an FPGA-accelerator direct access to user-space memory. The IOMMU serves its original purposes of protecting the host from malicious or faulty DMA devices and drivers [33] and of providing the DMA engine with the illusion of a physically contiguous buffer. As such, the host initiates the data movement from and to the FPGA memory through the Linux DMA API [14]: the IOMMU is set up as part of every DMA transfer preparation. This is not sufficient to support SVM between host and FPGA accelerator. Independent on if and when these issues might get resolved, the SVM framework we present provides an integrated, full-stack solution that abstracts away the low-level details and thereby enables a smooth SVM experience for application developers - also when using the hard-macro IOMMU in ZynqUS+.

Some recent research papers have highlighted that state-of-the-art IOMMUs might require modifications to address the needs of a particular target platform and/or application domain. Some point to the fact that a naive IOMMU configuration cannot meet the requirement of today's high-performance customized accelerators, as it lacks efficient TLB support (no flexibility whatsoever can be expected from such hard-macro IPs) [34]. The authors propose to leverage the host per-core MMU to implement the required additional functionality, as augmenting the hardware design to that aim would increase the complexity beyond what is affordable (this is especially true for low-end FPGA-based SoCs [35]). Other papers consider the problem of hardware complexity as a showstopper to implementing SVM in heterogeneous, low-end SoCs, and thus propose less intrusive solutions consisting of a simple hardware IOTLB that is completely software-managed by a kernel-level driver on the host [23] [21] [36]. The obvious downside of this approach is that the frequent interactions with the host and the associated interrupt latency can impose considerable overheads at accelerator run-time.

Focusing on IOMMU design for FPGA accelerators in embedded systems, a recent paper [37] highlighted that the best performance is achieved by a private, hardware-managed IOTLB. However, the speedup of roughly 12% compared to an IOTLB managed by software running on the host comes at the price of a 106% increase in logic resources with respect to the software-managed IOTLB. While the use of larger TLBs can also help to reduce the TLB service time, especially large and fully-associative TLBs quickly become costly in terms of resources [38] [39].

Alternative options for improving SVM performance at lower cost in FPGA-based heterogeneous embedded SoCs, such as modifying the architecture of the TLB itself, have not yet been widely addressed by the research community. The SVM framework presented in this work enables true SVM between host and FPGA accelerators by relying on a plug-and-play approach that allows the instantiation of a lightweight IOMMU in the FPGA. The framework does not require modifications to the host architecture nor does it restrict programmers to specialized memory allocators. Low-level details are taken care of without additional overheads at offload time. It supports coherent accelerator access to the data caches of the host as well as TLB prefetching transactions for improved performance. Unlike other designs providing similar capabilities, our design enables fine-grained control of such features for different address ranges.

3 Shared Virtual Memory Design

The FPGA accelerator is interfaced to the shared main memory interconnect through the input/output memory management unit (IOMMU), which translates virtual addresses as seen by the user-space application and the FPGA accelerator to the corresponding physical addresses in main memory. At the heart of any IOMMU design sits an input/output translation lookaside buffer (IOTLB). Hard-macro IOMMUs currently employed in high-end SoCs [13] [15] [16] additionally feature circuitry for TLB management like hardware page-table walking (PTW) engines.

However, the operation of the page-table walk itself is bound by the latency to main memory [22]. Moreover, despite some hardware IOMMUs are capable of directly and coherently operating on the host page tables, the Linux IOMMU API and the hardware drivers do not support this on ARM-based host systems [12].

Instead, a separate and empty I/O page table is generated at setup time. The first TLB miss to every page then generates a costly page fault that must be handled in software by the host by mapping the corresponding page to the I/O page table. The hardware management only helps for subsequent TLB misses on pages already mapped. Alternatively, all pages holding the shared data must be mapped at offload time, which is impracticable when operating on pointer-rich data structures. Finally, due to the decoupling of the I/O and the process' page table, the only way to ensure that the IOMMU does not use stale page-table data at any time is to prevent the mapped pages from being moved by page pinning, which further aggravates the cost for mapping and page fault handling.

Therefore, we do not consider hardware PTW and rely on a fully software-managed design for our soft IOMMU.

In the following, we present the two core components of our SVM framework. We start with the software infrastructure used for managing the IOMMU hardware and interfacing it with higher-level software and ultimately the application to be accelerated (Section 3.1). This component enables transparent SVM both when using a hard-macro IOMMU as available on some next-gen high-end FPGA-based SoCs (otherwise limited to simple operation in accordance with host-initiated DMA transfers) as well as when using the soft-core IOMMU that we provide (Section 3.2). Regarding this soft IOMMU, our focus is on studying parameterizable TLB design that allows for larger TLBs (thereby reducing capacity misses) and higher flexibility. Our design is highly configurable and efficiently uses the basic building blocks provided by today's FPGA devices. This not only enables SVM for FPGA accelerators on IOMMU-less SoCs, but it also proves beneficial when a hardware IOMMU is available. Due to higher flexibility, it can achieve better performance at negligible FPGA resource cost.

3.1 SVM Management

Fig. 1 visualizes how the hardware and the software layers of the framework interact. Once the application that needs to be accelerated is started, the runtime uses a system call to let the driver module register a kernel *worker thread* for handling TLB misses (page faults when using a full-fledged hard-macro IOMMU) with the *concurrency managed*

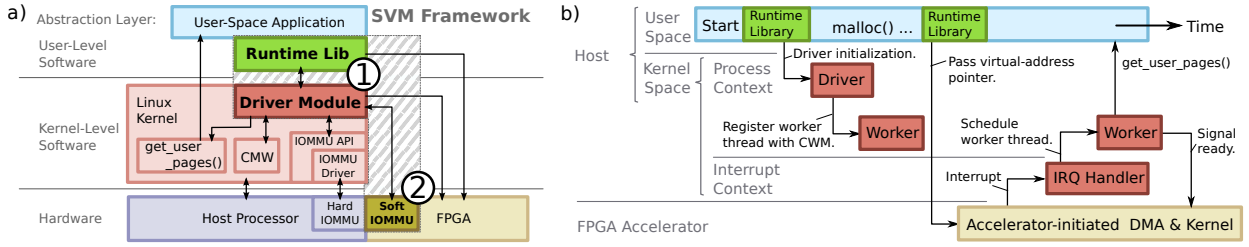

Fig. 1. a) Overview of the SVM framework with its two core components: software manager (1) and configurable soft-core IOMMU (2). b) Interaction of hard- and software components in different execution contexts during operation.

workqueue (CMW) API of Linux. The shared data elements can be allocated in memory by the user-space application as any other variable using the system's standard *malloc()* function. This allows for simple development and porting of already existing applications.

When offloading computation to the FPGA accelerator, the application developer just needs to specify the virtual address of the shared data elements which is then communicated to the accelerator, e.g., through a memory mapped mailbox. The accelerator can then access the shared data element using the virtual address pointer obtained from the runtime. In the case of a TLB miss or page fault, the interrupt handler inside the driver module simply triggers the execution of the *worker thread* in normal process context. Once this worker thread gets scheduled, it first reads the address and transaction attributes from the IOMMU hardware and pins the requested user-space page in memory using *get_user_pages()*. Then, it maps the pinned page to the I/O page table in case the hard-macro IOMMU is used, or performs virtual-to-physical address translation and sets up a new entry in the TLB if the soft IOMMU is used. If all TLB entries are in use, the oldest mapping is invalidated and the corresponding user-space page is unlocked (FIFO replacement). Based on the transaction attributes, the worker thread signals the hardware to repeat the transaction that previously produced a TLB miss/page fault [21].

In the case of a TLB miss that resulted from a prefetching transaction (marked in the transaction attributes obtained from the IOMMU hardware), no signal needs to be sent to the accelerator and the miss-handling thread can directly continue handling misses until the miss queue in the hardware is empty. The use of prefetching transactions allows the accelerator to request the setup of multiple TLB entries with a single TLB-miss interrupt, for example before setting up a DMA transfer touching multiple memory pages. This can improve performance as the miss-handling thread can handle multiple TLB misses in batch mode and is scheduled at most once, similar to the page fault handler of some IOMMUs found in high-end SoCs [17].

In addition, the runtime library allows the application developer to specify virtual address ranges and associate them with a specific TLB and an IOMMU master port (see Section 3.2) at offload time. The miss-handling thread then sets up new TLB entries within the specified range according to these settings. The library also allows for setting up the TLB statically at offload time and lock the corresponding TLB entries on a basis of shared data elements, if they can be completely remapped with the available TLB entries. Concretely, this allows fine-grained control of which address ranges (i.e., program data items) are to be looked up in the host cache and which ones are to be accessed directly in the dynamic random-access memory (DRAM). To this end, the runtime only needs the virtual address, size and access permissions, similar to the computation offloading directives of today's programming models for heterogeneous systems [40] [41].

3.2 IOTLB Design

Fig. 2 a) shows the block diagram of our soft-core IOMMU. The FPGA accelerator connects as master device to the slave port of the IOMMU, using the widely adopted AXI4 protocol on all its interfaces [42]. To enable the accelerator's reaction upon TLB misses (and to repeat the missing transaction once the IOMMU configuration has been updated by the host) we assume a similar accelerator wrapping methodology to [43], where a wrapper core provides the synchronization infrastructure on the accelerator side (See Section 4). The virtual address (VA) is fed to the control block together with meta information (length, transaction type and ID, AXI User signals) to perform a look-up of the VA in the TLB. A transaction that hits in the TLB is forwarded to the corresponding physical address in shared main memory. Based on the *master select* flag stored in the TLB, the transactions are forwarded to either of the two AXI4 master ports. The double data rate (DDR) port directly connects to the DRAM controller of the host CPU. The Accelerator Coherency Port (ACP) offered by FPGA-enabled SoCs [8] [9] allows the FPGA to directly access the most recent data copies from the host's data caches without the need for the operating system to flush the caches at offload time.[2] Read or write responses are directly forwarded from the downstream interconnect to the FPGA accelerator.

If a transaction misses in the TLB, its VA, ID and the AXI User Signals are stored inside the miss first-in, first-out buffers (FIFOs), and an interrupt is sent to the host CPU. The host then uses the AXI-Lite interface to read the miss FIFOs, and reconfigure the TLB accordingly (Section 3.1). In parallel, the IOMMU drops the transaction and signals a slave error in the AXI Read/Write Response back to the wrapper core inside the FPGA. The IOMMU does not block and can continue to handle address translations from other transactions to shared memory issued by the accelerator.

2. Next-gen SoCs [12] replace the ACP with more advanced ports supporting the ACE-Lite coherency extensions of the AXI4 protocol. Our IOMMU design is also compatible with ACE-Lite-enabled SoCs.

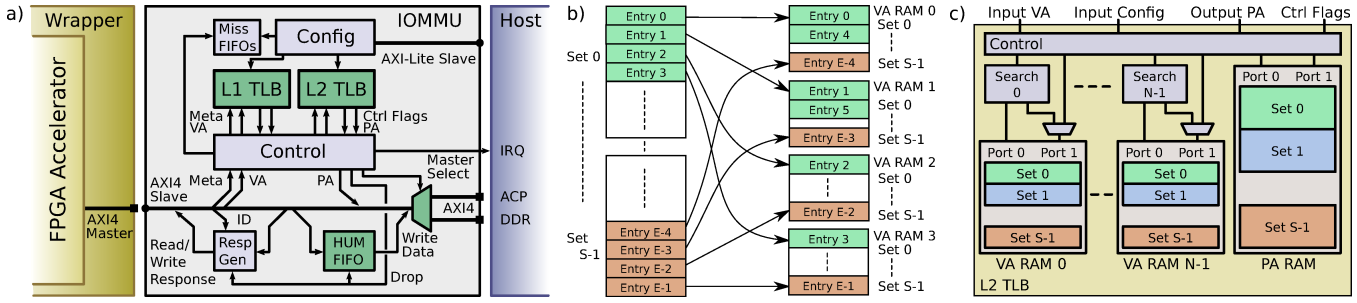

Fig. 2. a) Block diagram of the IOMMU, b) mapping of TLB entries to 4 VA RAMs, c) proposed L2 TLB architecture using dual-port BRAMs.

Using the AXI User signals, the accelerator also can mark transactions as TLB prefetches. In this case, the IOMMU performs the look-up in the TLB and signals back the result to the accelerator using the AXI Read/Write Response signals. However, the transaction is not forwarded to shared main memory in both the miss and the hit case. This allows the accelerator to request the setup of the TLB ahead of time without paying the latency to shared memory.

All AXI interfaces support configurable data and address widths. Also, the number of AXI4 ports is configurable, each of them having private TLBs.

TABLE 1
Hardware configuration parameters.

IOMMU	L1 TLB[a]	L2 TLB[a]
#Ports	#Entries	yes/no
Input Address Width		#Ways
Output Address Width		#Sets
Data Width		#VA RAMs
AXI ID Width		Page Size

[a] Configurable on a per-port basis.

3.2.1 Flexible, Single-Cycle L1 TLB

The first TLB design is optimized for low look-up latency and high flexibility. To this end, it is implemented using look-up table (LUT) and register slices of the FPGA [44] instead of block RAM (BRAM) hard macros [45]. It has a look-up latency of 1 clock cycle, is fully parallel and fully associative, and allows for arbitrary sized mappings, i.e., multiple memory pages can be remapped using a single TLB entry if they are contiguous in virtual as well as in physical memory. This allows to efficiently support techniques targeting at reducing the host TLB miss rate such as transparent huge pages [3] and contiguous physical memory obtained, e.g., through the CMA [4]. Low TLB look-up latency and high associativity are important whenever virtual-to-physical address translation is in the critical path, such as for CPUs where every memory transaction has to be translated. However, such designs do not scale well to larger sizes and quickly become very costly in terms of resources [38] [39]. In practice, they are limited to sizes up to 64 entries.

3.2.2 Set-Associative, Multi-Cycle L2 TLB

To overcome the limitations of traditional fully-parallel, fully-associative FPGA TLB designs [24] [32] [39], we propose a new and scalable TLB architecture for FPGAs. It relies on sequentially searched BRAM to allow for high-capacity TLBs and thus reduced overall TLB service time at reasonable FPGA resource cost. To achieve look-up latencies comparable to fully-parallel, pipelined TLB designs, our architecture 1) is n-way set associative, 2) parallelizes the look-up using multiple dual-port BRAM cells, and 3) starts the look-up at the position of the last TLB hit, which brings the effective look-up latency down to the minimum of 3 cycles for bandwidth critical workloads. The maximum look-up latency is

$$L_{\text{max}} = 2 + \frac{n_{\text{ways}}}{2 \cdot n_{\text{RAMs}}}$$

where n_{RAMs} is the number of BRAMs searched in parallel. Typically, L_{max} is between 4 to 18 clock cycles. How the sets are distributed over the parallel RAMs is shown in Fig. 2 b). The mapping of VAs to sets is based on the least-significant bits (LSBs) of the page frame number (PFN). In contrast to the flexible L1 TLB, this design is restricted to page-sized mappings, but its architecture is highly configurable at compile time. Table 1 lists the configuration parameters.

The architecture of this TLB is visualized in Fig. 2 c). Upon receiving an input VA and a start signal from the top-level control block of the IOMMU, the set index is determined and forwarded to the parallel search units together with the virtual PFN. Every search unit is connected to a dual-port BRAM cell holding the VAs (VA RAMs). Per cycle, every search unit reads two entries of the current set from the VA RAM (using both ports of the BRAM cell), and compares the PFNs stored in these entries with the PFN of the input VA. If a matching entry is found, its index is forwarded to the control block together with a hit signal. The control block aborts the search, reads the corresponding physical PFN from the single physical address (PA) RAM, and then outputs the PA together with control flags to the top-level control block. The offset of the matching entry is stored in a set-specific register in the control unit. The next search in the set is then started at this position to speed up the search and exploit possible locality of reference.

In case the entire set is searched without finding a valid entry or if a valid entry is found that does not allow the required transaction type, the search is aborted and a corresponding signal is sent back to the top-level control block. The VA RAMs are configured through Port 1.

3.2.3 Hybrid TLB

A real system can benefit from a combination of the two TLB designs to form a *hybrid* architecture. A small L1 TLB

3. Refer to https://lwn.net/Articles/423584/
4. Refer to https://lwn.net/Articles/486301/

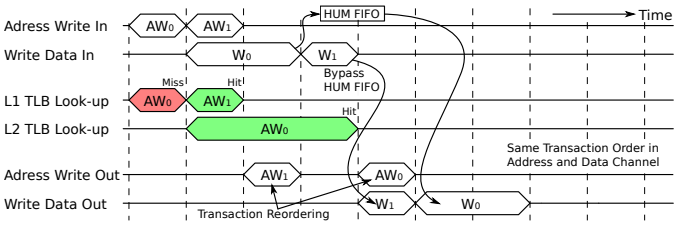

Fig. 3. Hit-under-miss (HUM) support in the write channel.

with 4 to 8 variable sized entries, e.g., to efficiently support techniques designed to reduce host TLB misses such as CMA, and a large L2 TLB with higher look-up latency and page-sized but much cheaper entries for regularly allocated memory pages. Thus, our design supports the instantiation of both TLB blocks.

To allow for parallel look-ups of the two TLBs in the hybrid architecture, our design supports the hit-under-miss (HUM) feature, which can lead to reordering of transactions. To ensure correct ordering in both the AXI Write Address and Write Data channels, typical hardware IOMMUs [13] buffer the entire write data burst, which can require a considerable amount of buffer memory (an AXI4 burst can comprise up to 256 data beats). Instead, our IOMMU just uses a FIFO buffer with the size equal to the maximum look-up latency of the L2 TLB (HUM FIFO in Fig. 2 a)). As visualized in Fig. 3, for a missing transaction AW_0 with a burst length lower than or equal to the L2 TLB latency, the data W_0 is buffered by the HUM FIFO. In the meantime, Transaction AW_1 hits in the L1 TLB and is fed to the output. Once the associated data W_1 arrives at the input, it bypasses the HUM FIFO and is directly fed to the output. When the physical address for AW_0 has been found in the L2 TLB, the transaction is fed to the output and data W_0 is retrieved from the FIFO.

If the burst length of the transaction missing in the L1 TLB is larger than the L2 TLB latency, there is no reordering. Even if the next transaction hits in the L1 TLB while the L2 TLB is busy, there is little benefit only in forwarding it downstream – independent of the buffering capability. The corresponding write data will only arrive at the input after the L2 TLB look-up is finished (due to the burst length of the missing transaction) and write-data interleaving is not allowed in AXI4. If a transaction misses in the L1 TLB while the L2 TLB is busy, the input address channels are stalled until the L2 TLB becomes available again.

Note that our SVM framework does not block the accelerator's traffic to shared memory in case of outstanding TLB misses, independently of the selected TLB architecture and organization. The purpose of the HUM feature is just to allow for parallel look-ups if both TLBs are being instantiated inside the IOMMU.

4 Evaluation Platform

To evaluate the performance of our SVM framework and explore the different TLB designs in a real system, we set up two platforms: one based on the Xilinx Zynq-7000 SoC [8] and one based on the Xilinx Zynq UltraScale+ MPSoC [12]. For simplicity, we will refer to them as *Zynq* and *ZynqUS+* in the following. Fig. 4 shows the Zynq platform.

TABLE 2
Effective and relative FPGA resource utilization of the IOMMU configuration selected for the experimental evaluation.

Sub Block	LUTs	FFs	BRAM
L1 TLBs [a]	6.63 k	4.67 k	0.00 kbit
L2 TLB [b]	0.29 k	0.14 k	45.05 kbit
Buffers & Control	1.71 k	2.75 k	1.09 kbit
Total	8.64 k	7.56 k	46.14 kbit
Rel. Zynq-7020	16.24 %	7.11 %	0.94 %
Rel. Zynq-7045	3.95 %	1.73 %	0.24 %
Rel. Zynq US+ 9EG	3.15 %	1.38 %	0.14 %

[a] 32 entries (accelerator to host), 4 entries (host to accelerator)
[b] 1024 entries (4 VA RAMs with 32 sets, accelerator to host)

Both SoCs feature an ARM Cortex-A host CPU running Xilinx Linux 4.9. On Zynq, this is a dual-core A9 whereas ZynqUS+ uses a quad-core A53. The coherent interconnect has additional slave ports, which allow hardware blocks implemented in the programmable logic to access the data caches inside the CPU. Zynq uses ACP whereas ZynqUS+ uses a more advanced ACE-Lite port. For simplicity, we refer to the coherent port as ACP both for Zynq and ZynqUS+ in the following. In case the requested data is not held in cache, the transaction is sent to main memory using the DRAM controller of the host. The system features 1 GiB of DDR DRAM, which is shared between host and FPGA.

To interface the programmable logic of the with the host, we instantiate an IOMMU configuration with two ports. The first port is used for host-to-accelerator communication and uses an L1 TLB with 4 entries only. The second port is used for accelerator-to-shared-memory communication. It features two master ports: One connected to the DDR DRAM controller of the host and one connected to the ACP. Also, it is used to instantiate both TLB designs. The L1 TLB is instantiated with 32 entries, i.e., the maximum size for which clock frequencies above 100 MHz are achievable on Zynq (see Section 5.1). The L2 TLB is instantiated with 4 parallel VA RAMs with 32 sets and 1024 entries in total, and a max look-up latency of 6 cycles.

In addition, ZynqUS+ features an ARM SMMU, i.e., a full-fledged, hard-macro IOMMU [13] that can be leveraged for SVM using the software components of our framework. The SMMU has 6 ports with a private, fully-associative L1 TLB each. These ports connect to a shared management unit through an AXI Stream interface. The management unit features a shared L2 TLB, a multi-thread hardware PTW engine, PTW caches and prefetching buffers. The SMMU is clocked at 533 MHz. In both platforms, the host is configured to run at 666 MHz and the FPGA at 100 MHz.

Table 2 gives the absolute FPGA resource utilization of the IOMMU configuration used for the evaluation on Zynq. The table also gives the relative utilization on a small- and medium-sized, mid-end device, as well as on a medium-sized, high-end UltraScale+ device. The selected configuration uses less than 4% of the available resources on the medium-sized devices and leaves plenty of space for actual accelerators.[5]

5. For example, the implementation of an accelerator for sparse matrix-vector multiplication [46] would use roughly 60% of the LUTs and FFs, and 57% of the BRAMs offered by the Zynq-7045 device.

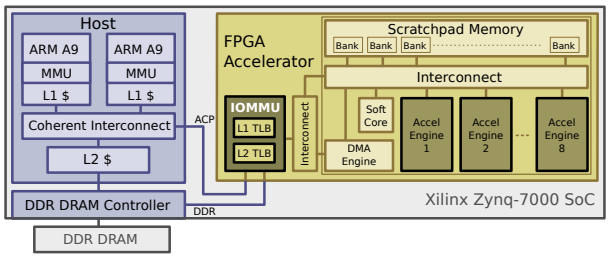

Fig. 4. Zynq evaluation platform with FPGA accelerator featuring 8 parallel accelerator engines, SPM, DMA engine and soft core for control.

Attached to the IOMMU, there is a programmable FPGA accelerator [43]. Eight *accelerator engines* can directly access a local SPM (256 KiB) and the main memory via both single-word loads/stores and more efficient DMA burst accesses. The accelerator uses a soft core for higher-level control tasks. These include the configuration of the accelerator engines, the management of DMA transfers between scratchpad and main memory, and the synchronization with the host. This core can also be used to ensure that the corresponding TLB entries are set up in our IOMMU. This is achieved by issuing prefetching accesses to the memory pages touched by the DMA transfer and evaluating the response returned by the IOMMU [21]. In the case of a miss, it goes to sleep and waits for the host to set up the TLB. After handling the miss, the host wakes up the soft core that generated the miss. To avoid cache pollution by the accelerator when accessing the shared memory through the ACP, DMA transactions are not allocated in the data caches of the host in the case of cache misses. The maximum DMA burst size is 256 B, and the accelerator's peak bandwidth to main memory is 6.4 Gb/s.

The accelerator engines generate the memory access patterns of four real applications, which have been parameterized to capture the effect of varied input datasets. A brief description of these application kernels follows.

Pointer Chasing (PC): This kernel is characteristic of a wide variety of pointer-based applications such as PageRank, breadth-first search, shortest path search, clustering, and nearest neighbor search [47]. The host builds up a linked list in virtual memory. Every list element represents a graph vertex and stores the number of successors, a pointer to a list of successor-vertex pointers, and a configurable amount of payload data. The host passes a pointer to the list to the FPGA accelerator, which then starts traversing the graph. If the number of successor vertices is greater than zero, it sets up DMA transfers to copy the payload data and the successor pointers to local memory. Then, it does a configurable number of computation cycles, and writes the payload data to the successor vertices in shared main memory again using DMA. The traversal is parallelized on a vertex level, leading to parallel, independent DMA streams.

Random Forest Traversal (RFT): This kernel is typically used for pattern recognition as well as regression and classification algorithms based on binary decisions trees [48], [49]. The host generates regular decision trees of configurable size in virtual memory and passes pointers to the root vertices as well as to the input samples to the FPGA. The accelerator classifies every sample with every tree. The decision trees are stored using a large, 1-dimensional array. For every vertex, the array holds the limit for the binary decision as well as a configurable amount of payload data. When a sample arrives at a vertex, the accelerator first loads the payload data using DMA and then performs a configurable number of computation cycles. Based on the result of this computation, the sample is passed either to the left or to the right successor vertex. When arriving at a leaf vertex, the corresponding index is returned to the host.

Because of irregular, data-dependent memory access patterns with low locality of reference, PC and RFT represent worst-case scenarios for (virtual) memory systems. However, it is for such pointer-rich kernels that SVM is so beneficial as it reduces design time efforts for heterogeneous implementations. Without SVM, programmers must rethink communication patterns, data structures and their management on the host, which results in complete redesigns of whole applications instead of just porting critical kernels to the FPGA. That said, such kernels are effectively accelerated using FPGAs thanks to high degrees of parallelism [1], [2].

Sparse Matrix-Vector Multiplication (SMVM): This is a typical example application amenable to FPGA acceleration. The host encodes a potentially very large, sparse matrix in Condensed Interleaved Sparse Representation (CISR) [46] and then passes virtual address pointers to the matrix as well as to the input and output vector to the accelerator. The CISR format statically schedules the computation to 8 parallel accelerator pipelines on a row basis, thereby allowing for load balancing at low communication overhead and resource cost [50]. The accelerator uses DMA transfers to first fetch the dense input vector and then continuously streams in the sparse matrix.

Memory Copy (MC): This application's access patterns to shared memory are highly regular. The host allocates a buffer of configurable size in virtual memory and then passes a virtual address pointer to the FPGA. The FPGA accelerator uses DMA transfers to copy the buffer from shared memory into the local SPM at maximum bandwidth. MC is a representative example for streaming applications with low operational intensity.

5 Results

5.1 Resource Utilization

To evaluate the resource utilization and performance of the TLB designs we have implemented various configurations for a XC7Z045FFG900-3 device using Xilinx Vivado 2016.3. The system uses 32-bit address width, 64-bit data width and a page size of 4 KiB.

Fig. 5 a) compares the resource utilization and the TLB look-up time for different configurations of the two TLB designs when targeting maximum clock speed. For the flexible, single-cycle L1 TLB design we varied the number of entries. The resource utilization of the L1 TLB increases linearly with the number of entries. The same holds for the longest path delay, as the comparator network for performing the single-cycle TLB look-up does not map well to FPGA logic.

The multi-cycle L2 TLB design outperforms the L1 TLB in terms of resource utilization and longest path delay. For example, using the L2 TLB design to build a fully-associative TLB with 32 entries allows to reduce the logic resource utilization by more than 17x compared to the L1

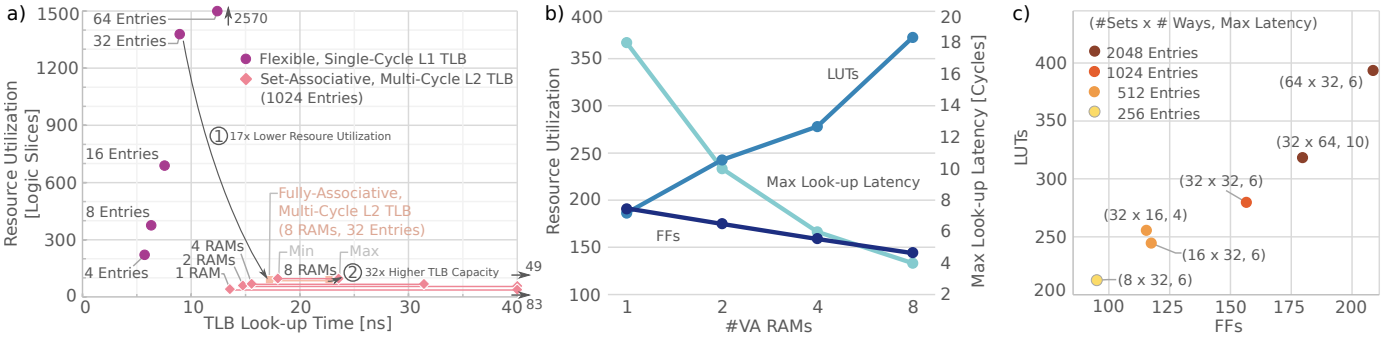

Fig. 5. a) TLB resource utilization vs. look-up time when optimizing for maximum speed. Resource utilization and max look-up latency of the set-associative, multi-cycle L2 TLB design for b) different numbers of parallel VA RAMs and c) different sizes and associativity with 4 VA RAMs.

TLB (1). This comes at the price of a larger and variable TLB look-up latency and the restriction to page-sized mappings. The minimum look-up latency for this TLB is 3 clock cycles whereas the maximum latency depends on the number of parallel VA RAMs. The longest path runs from the output of the VA RAMs through the search units into the PA RAM. The main advantage of this design is that it enables the construction of high-capacity TLBs at low resource cost. As shown in Fig. 5 a), switching from a fully-associative to a 32-way set-associative configuration allows for a 32x increase in TLB capacity at an increase in logic resource utilization and longest path delay of just 20% and 4%, respectively (2).

By varying the number of parallel VA RAMs, the design of the L2 TLB allows for a trade-off between maximum look-up latency in cycles and resource utilization while keeping the number of entries constant. This trade-off is also visualized in Fig. 5 b) for the same configuration with 1024 entries divided into 32 sets. As shown in the figure, the number of slice LUTs increases logarithmically with the number of VA RAMs whereas the number of slice flip-flops (FFs) decreases with more VA RAMs. The reason for this decrease in FFs is that as the number parallel VA RAMs increases, the number of bits required to store the offset of the last hit decreases. Using 4 parallel VA RAMs offers a good trade-off between maximum look-up latency of 6 cycles and logic resource utilization. Note that, as the number of parallel VA RAMs increases from 1 to 4, the utilization of the individual BRAM cells decreases by a factor of 4x from 75% to 19%.[6] How the logic resource utilization with 4 VA RAMs changes with the number of sets and the number of ways (set size) is shown in Fig. 5 c). The configuration using 32 sets and 32 ways (1024 entries) offers a good trade-off. For TLB sizes greater than 1024 entries, either the logic resource utilization or the maximum look-up latency increases sharply.

5.2 Microbenchmarks & Profiling

5.2.1 *Software Primitives for SVM Management*

We have evaluated the cost of the software primitives of our design using a synthetic benchmark. The host passes a pointer to an array in virtual memory to the accelerator. The accelerator reads the pointer and performs accesses to the array. Using performance counters inside the host and the accelerator, the duration of the various phases during miss handling can be profiled. The results are shown in Table 3.

6. Xilinx 7 Series devices feature BRAM cells with fixed dimensions of 16 or 32×1024 bit (not including parity bits) [8].

TABLE 3
Average delay in host cycles for sharing a single 4 KiB memory page.

	Response	Schedule	Handling	Total
TLB Miss, DDR	4,400	9,900	14,000	28,300
TLB Miss, ACP	4,400	9,900	8,900	23,200
Copy Page Out to Shared Memory				43,500
Copy Page In to Virtual Memory				87,500
Copy Page From SVM to SPM, DMA Read				5,100
Copy Page From SPM to SVM, DMA Write				4,400

The average cost for handling a TLB miss is 28,300 cycles when the accelerator accesses the data from DRAM. Using the ACP avoids the need to flush the host's data caches and allows for 18% faster miss handling. Most of the time is spent on waiting for the kernel worker thread to get scheduled (51 or 62% for DDR and ACP, respectively). This is required as the routine walking the page table and pinning the pages uses a kernel API which may sleep and, therefore, cannot be executed in interrupt context.

Copying a 4 KiB memory page of data from virtual memory to the physically contiguous, uncached memory reserved for the accelerator in case the platform does not support SVM takes 43,500 host cycles. Copying back a page that the accelerator has modified costs at least 87,500 cycles.

While the implementation of a routine to copy raw data is straight forward, the traversal and translation step in case the shared data contains virtual address pointers is application dependent and left to the programmer. The associated design time overheads are high and usually require the programmer to completely rethink and restructure the application. Also, the traversal and translation step incurs a big run-time overhead at offload time, which is much larger than that of copying raw data. However, once the costly offload including the data copying has been done, the accelerator's accesses to the shared data come at no overhead. In contrast, the cost for setting up a TLB entry have to be paid on every miss to a particular page. Thus, if data reuse is high, copy-shared memory can out-perform SVM. This will be highlighted by the exploration in Sec. 5.3.

For maximum performance, the accelerator operates on data residing in its local SPM. This memory is physically addressed and managed by the accelerator itself, which uses high-bandwidth DMA transfers to copy data between

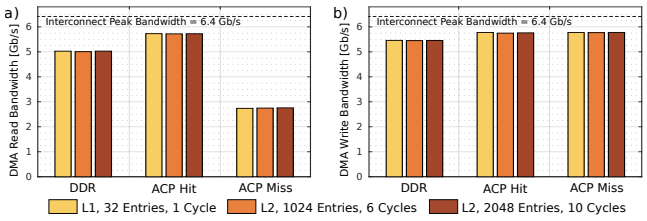

Fig. 6. Maximum DMA bandwidth to SVM for a) read and b) write transfers of 32 KiB when using different TLB configurations.

SVM and the SPM. The latency of setting up a single DMA transfer is 6 accelerator-clock cycles. Table 3 also lists the full cost for transferring a single 4 KiB memory page between SVM and the SPM using the DMA engine.

5.2.2 Hard-Macro IOMMU Profiling

A similar microbenchmark was designed to determine unspecified architectural key parameters of the hard-macro IOMMU available on the Xilinx UltraScale+ MPSoC [13] and profile the management primitives.

The per-port private, fully-associative L1 TLBs have a size of 64 entries and a look-up latency of 1 clock cycle (at 533 MHz). The shared L2 TLB has a size of 512 entries and a look-up latency of 39 cycles. This also includes the communication latency from the per-port L1 TLB to the shared management unit over the shared AXI Stream interface. Thanks to the hardware PTW engine, prefetch buffers and PTW caches, handling a TLB miss takes just 31 clock cycles on average (not including TLB look-up latencies). However, this only holds for capacity misses. The first access to every page leads to a compulsory TLB miss and a page fault as the requested user-space page has not yet been mapped to the I/O page table traversed by the hardware PTW engine. To handle this page fault, the host needs to pin the user-space page in memory and create the corresponding mapping in the I/O page table. This takes 12,400 host clock cycles on average (90,400 cycles max). During that time, the IOMMU completely blocks any memory transactions.

In contrast, our own IOMMU design allows to overlap TLB miss handling on the host with actual IOMMU operation. In addition, multiple TLB misses can be handled in batch mode, which can bring down the effective TLB miss latency as the host needs to handle a single interrupt only and schedule the miss handling routine just once.

5.2.3 DMA Peak Performance

To evaluate possible effects of the TLB configuration and look-up latency on the maximum DMA bandwidth, we used another synthetic benchmark that lets the accelerator issue DMA transfers with the maximum size of 32 KiB. The TLB is statically set up for this measurement and the host is idle during the measurement. As shown in Fig. 6 a), the highest bandwidth for read transfers (90% of the peak bandwidth of the interconnect) is achieved when using the ACP and if the requested data is in the data caches of the host (ACP hit).

If the data is not in the caches (ACP miss), the effective bandwidth is substantially reduced as the data has to be retrieved from DRAM using the same physical interconnect used by the host. In case the host is heavily loaded, the traffic injected by the accelerator can lead to additional contention and performance degradation. Compared to using the DDR port, the achievable bandwidth is roughly 45% lower. Letting the FPGA coherently access data from the caches of the host is not always beneficial.

For write transfers, the maximum bandwidth is between 85 and 90% of the peak bandwidth as shown in Fig. 6 b).

The TLB configuration and maximum look-up latency do not have an impact on the maximum bandwidth. This is a result of the proposed L2 TLB design starting the look-up at the position of the last TLB hit.

5.3 Real Traffic Patterns

In this section we evaluate the performance of the SVM framework on ZynqUS+ when using our IOMMU and the various TLB designs (*L1*, *L2*, *hybrid*) as well the hard-macro IOMMU (*SMMU*). The accelerator is used with eight engines and under various workload conditions (Table 4). The selected parameters represent a mix of real use cases [47] [48] [49] [51], problem sizes suitable for embedded systems and performance crossover points of the system.

For *Pointer Chasing (PC)*, we used an Erdős-Rényi graph [52] with 10 k vertices (input data is randomized). During our experiments, we found that graphs with different vertex count do not lead to big differences in the results as long as the total graph size is larger than the TLB capacity.

Random Forest Traversal (RFT) operates on random input numbers which are sorted by the tree. Therefore, the obtained access patterns to shared main memory are highly irregular and randomized, and the benchmarks represent a worst-case scenario for an SVM sub-system.

Comparing with copy-based shared memory, data reuse matters. Therefore, we varied the number of input samples fed to RFT as well as the number of iterations performed in PC. The number of iterations performed varies between 1 and 6 for most applications, while some require up to 70 [47]. In many cases, the execution is stopped after 5 iterations, where 95% of the vertices have converged [51].

For *Memory Copy (MC)* we consider as main parameters the size of the data chunks (64 KiB and 1024 KiB) and the number of compute iterations executed on the offloaded data (data reuse). For *Sparse Matrix-Vector Multiplication (SMVM)*, we have used four matrices with different sizes and numbers of non-zero entries from the University of Florida Sparse Matrix Collection [53], as shown in Table 4.

TABLE 4
Benchmark parameter sets.

Pointer Chasing		Random Forest Traversal	
#Vertices	10 k	#Tree Levels	4 - 16
Vertex Size [B]	44 - 2,060	Vertex Size [B]	28 - 268
#Cycles per V.	10 - 10,000	#Cycles per V.	10 - 1,000
#Iterations	1 - 64	#Input Samples	256 - 2,048

Sparse Matrix Vector Multiplication [53]				
Matrix Name	#Rows	#Cols	#Non-Zero Entries	Data Size [KiB]
power	4,941	4,941	13,188	180
ca-HepTh	9,877	9,877	51,996	561
Dubcova1	16,129	16,129	269,138	2,355
olafu	16,146	16,146	1,031,302	8,309

We have measured the accelerator run time (including offloading time) for the different schemes and normalized results to the copy-based memory sharing, the state of the art in embedded heterogeneous systems. The accelerator uses prefetching transactions before setting up DMA transfers touching multiple pages. This improves performance as the host needs to handle at most one TLB-miss interrupt and schedule the miss-handling thread at most once per DMA.

With our design, the application developer has fine-grained control of the IOMMU settings on the basis of virtual address ranges and/or data elements. For example, this allows to associate a specific address range with the *L2* TLB and the DDR port for maximum bandwidth, while the ACP is used for other shared data elements. As MC and PC both share a single data element only, we have evaluated the different settings in isolation without the *hybrid* TLB configuration for these two benchmarks.

In contrast, the data sizes for RFT and SMVM quickly exceed the size of the data cache of the host. This reduces the efficiency of DMA transfers on the ACP. Thus, we always used the DDR port only for these two benchmarks.

5.3.1 Pointer-Chasing (PC)

Fig. 7 shows the performance of *PC* (different curves represent run-time for different TLB schemes), normalized to the performance of the baseline copy-based shared memory. The x-axes of the plots show the total graph size when increasing the vertex size from 44 B to 2 KiB. The two leftmost plots refer to a configuration where for each vertex loaded from memory, 10 cycles are spent doing computation. For the two rightmost plots the computation cycles per vertex are 10 k. Typical *PC* applications perform 1 to 6 iterations on the graph [47], [51], thus we report results for 1 iteration (plots *a* and *c*) and 4 iterations (plots *b* and *d*).

For a single iteration, using the L2 TLB allows to achieve speedups of up to 2.2x. L2 DDR performs better for graph sizes below 4 MiB, i.e., when the entire graph can be remapped by the L2 TLB and the execution is thus bandwidth limited. As the graph size increases, the number of TLB misses increases. The execution time becomes more dominated by handling capacity misses in the TLB, while the ACP performs better as no cache flushes are required.

As the number of iterations on the graph increases, the offload cycles in the case of copy-based shared memory are less predominant. The relative performance of L2 DDR and L2 ACP deteriorates for graph sizes above the TLB capacity. In contrast, the SMMU performs better for larger graph sizes. The multi-threaded hardware PTW can deal with the increasing number of capacity misses in the TLB. For small graph sizes, the performance of the SMMU is dominated by page fault handling. Upon the first access to every page, the SMMU interrupts the host, which must map the requested page to the SMMU's page table. During that time, the SMMU blocks any traffic which prevents exploiting the full potential of the SMMU hardware.

As the number of compute cycles per vertex increases to 10 k, the relative speedup when using the L2 TLB decreases as shown in Fig. 7 c) and d), respectively. Due to the increased operational intensity, the time spent for memory transfers accounts for a smaller portion of the total run time. **For larger vertex and graph sizes, more computation cycles help to overlap the miss-handling time with useful computation.** In contrast, copy-based shared memory does not allow to overlap the costly offload procedure with accelerator computations. The relative performance saturates at around 60% even when using the much smaller L1 TLB.

It is important to underline that, while SVM can perform worse than copy-based memory sharing in some scenarios, its programmability is always much better. This is particularly true for *PC*, where running the copy-based approach required a complete rewrite of the original program.

5.3.2 Random Forest Traversal (RFT)

Fig. 8 a) and b) plot the performance versus increasing tree depth for different vertex sizes but a fixed operational intensity of 0.2 cycles/Byte. Since during the offload of this application only few pointers need to be adjusted and since the accelerator only performs read accesses to the data structure, already little data reuse amortizes the offload cost. In contrast, the access pattern to the tree is highly irregular. This creates many capacity misses.

Due to the irregular pattern and the small vertex size, there is little temporal locality, which unveils the higher maximum look-up latency of the L2 TLB compared to the L1 TLB in a). **The relative performance increases for growing tree depths where the higher look-up latency of the L2 TLB is compensated by the higher overall hit rate.**

Larger vertex sizes let SVM become more beneficial as they amortize the miss handling cost (more data is accessed

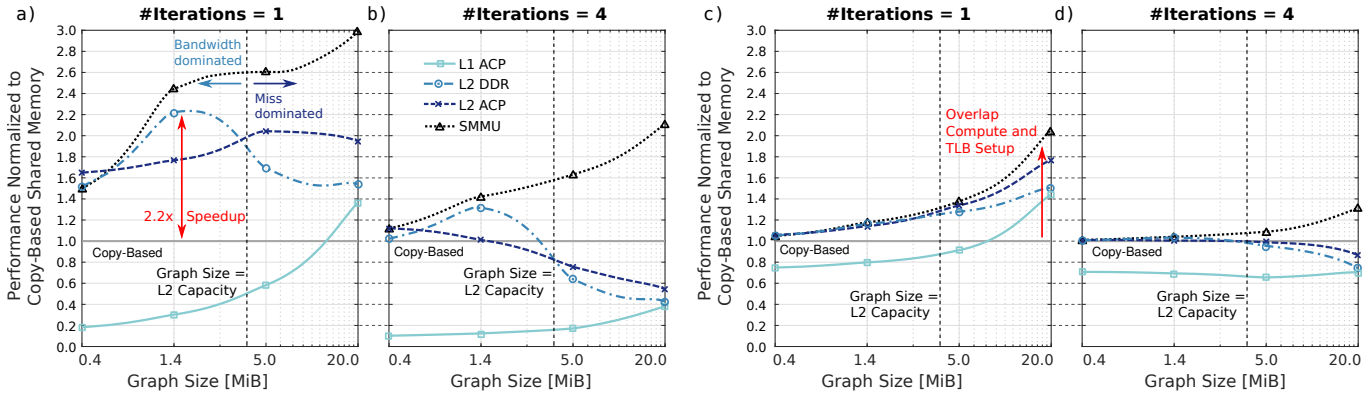

Fig. 7. PC performance for varying vertex/graph size, number of iterations: 10 cycles per vertex in a) and b), 10,000 cycles per vertex in c) and d).

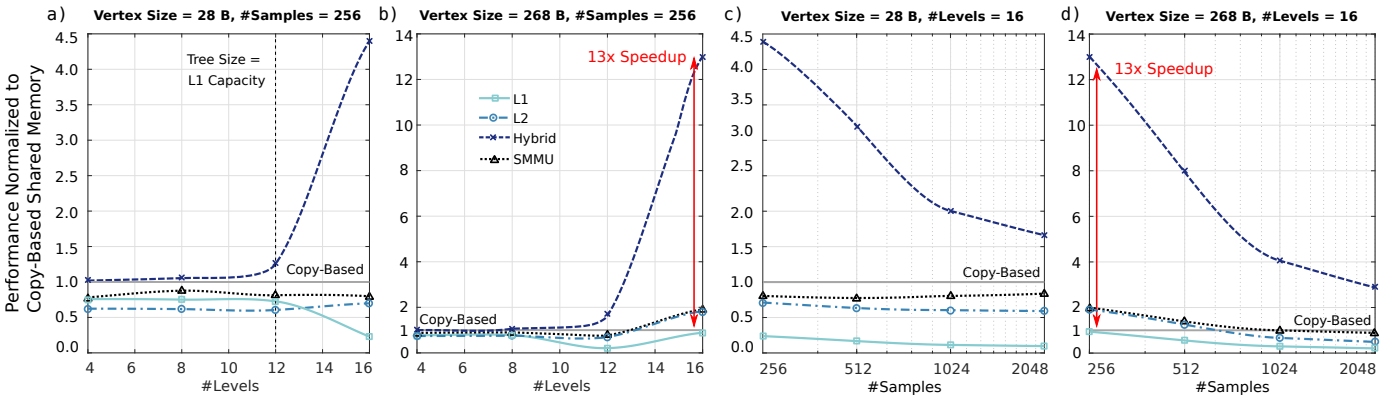

Fig. 8. RFT performance for an operational intensity of 0.2 cycles per Byte when varying the tree depth a) and b), and when varying the number of samples fed to the tree c) and d).

per miss). This effect increases with the tree depth as on the lower levels, almost every access to the tree is a miss. Also, a lot of data is copied when using copy-based shared memory that is potentially never accessed by the accelerator. This can lead to very high speedups of up to 13x as shown in b).

The speedup decreases as the number of samples fed to the tree increases as shown in Fig. 8 c) and d). **The highest performance is achieved when using the hybrid design and allocating the trees in physically contiguous memory. Accesses to the trees then always hit in the L1 TLB.** However, the hybrid design is not suitable for on-line learning [49] as the host needs to regularly access the physically contiguous, uncached memory to update the trees. In this case, the L2 TLB only or the SMMU should be used. It performs better for up to 768 samples (tree updates usually have to performed earlier).

5.3.3 Mem Copy (MC)

Fig. 9 shows the performance of MC normalized to copy-based shared memory for a data size of 64 KiB and 1024 KiB. The maximum speedup is 2.2x (L2 ACP). As the size of the SPM might not suffice to hold all the required data for a specific accelerator kernel, multiple iterations over the same input data might be required, e.g., to apply a set of filter kernels on a given input image. The x-axes of the plots denote the number of iterations performed on the data.

There are multiple reasons for the drop in relative performance with increasing number of iterations. First, in the case of copy-based shared memory, more accesses to the copied data amortize the initial offload cost. Second, with more iterations, the execution time becomes less dominated TLB misses but by the effective main memory bandwidth if the data size is below the TLB capacity. Using the ACP should be avoided in this case due to the lower bandwidth resulting from cache misses and contention. Fig. 9 a) shows that while the ACP gives the best performance for few iterations, the performance using the ACP saturates at 60% of the DDR port in the bandwidth dominated regime.

The L1 TLB performs equally well as the L2 TLB, but it does not outperform it, despite the lower look-up latency. The FPGA uses latency-insensitive DMA transfers to access main memory. The actual computations are performed on local SPMs for which address translation is not required. **Unlike for CPUs, the TLB is not in the critical path for FPGA accelerators. Instead of optimizing the TLB for low look-up latency, the available FPGA resources are better invested in building a TLB with larger capacity.** As the data size and the number of iterations increase (Fig. 9 b), only the configuration using the large L2 TLB and the DDR port can compete with copy-based shared memory. The performance of the SMMU is dominated by the interrupt latency due to page fault handling. While the host is handling an IOMMU page fault, the SMMU completely blocks any traffic. In contrast, our design allows multiple outstanding misses to be handled using a single host interrupt.

5.3.4 Sparse Matrix-Vector Multiplication (SMVM)

Independent of the problem size, the performance with our design is at least 1.5x higher compared to copy-based shared memory as shown in Fig. 10. SMVM features a linear access pattern to shared memory. Every input and output data element is read and written exactly once by the accelerator, respectively. Only compulsory TLB misses happen and the speedup is proportional to the cost ratio between handling a TLB miss and copying a memory page between contiguous and virtual memory. The only benefit of a larger TLB (L2) is that it allows for larger DMA transfers, which leads to a slight increase in performance.

The use of a hybrid configuration leads to a significant increase in performance. In such a configuration, the matrix

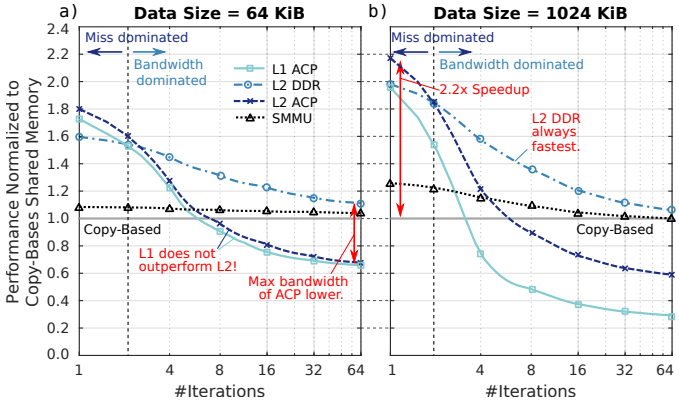

Fig. 9. MC performance for different number of iterations and data sizes.

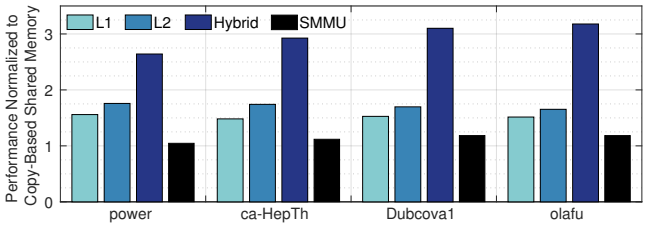

Fig. 10. SMVM performance for different sparse matrices.

is allocated in a virtually and physically contiguous memory region remapped using a single but variable sized entry in the L1 TLB, and where the input and output vectors are remapped using the page-sized entries of the L2 TLB. Accesses to the matrix thus never miss in the L1 TLB. Since for many practical applications of SMVM, the same matrix is multiplied many times with varying input data and exclusively accessed by the accelerator, it pays off to let the host once place the matrix in a contiguous but uncached memory region, e.g., obtained through CMA. In contrast, the input and output vectors also processed by the host but accessed only once by the accelerator should be allocated in normal, cached memory remapped using the L2 TLB.

The performance of the SMMU is dominated by handling page faults. Similar to MC, the access pattern is linear and every data element is accessed once. There are only compulsory but no capacity TLB misses. This does not allow to amortize the cost for the page faults and prevents the exploitation of the SMMU hardware.

5.4 Comparison with Related Works

5.4.1 Hardware Comparison with Related Works

Table 5 compares relevant IOMMU FPGA implementations reported in the literature with a hybrid configuration of our design featuring 4 variable-sized L1 TLB entries and an L2 TLB (4 parallel VA RAMs, 1024 entries, 32 ways). The maximum clock frequency of this configuration is 185 MHz. To further increase the performance of our design, e.g., when used to interface an FPGA accelerator running at faster clock speeds or that saturates the available bandwidth, our design can still be adapted by adjusting the data width of the AXI4 interfaces which is a configurable design parameter.

Only three of the considered FPGA IOMMUs run faster than our design, whereas in terms of effective look-up time and relative latency compared to the corresponding host ours is the best. The software-managed MMU for soft processors from Shamani et al. [39] is comparable to our design in terms of FPGA technology, speed and resource utilization. Its two TLB levels are fully-associative and implemented using fully-parallel, pipelined FPGA logic. This enables high clock speeds but makes the TLB resource hungry[7] and limits it to substantially smaller capacities. To achieve high clock frequencies, the two fastest designs both implement a (heavily) pipelined TLB using BRAM cells. This leads to a higher resource utilization and a look-up latency which is comparable [32] or much higher [24] than that of our IOMMU, despite the much smaller TLB capacity.

Using private, HLS-generated translation hardware for every shared data element [27] might lead to a better resource utilization in some cases, but this clearly heavily depends of the target application. In the worst reported case the use of resources is higher than ours, for a design that runs significantly slower. The pipelined page-table walker engine from Winterstein et al. is equipped with two intermediate TLBs and a physically-addressed data cache [54]. This RTL module enables SVM for HLS accelerators described in OpenCL. It uses more resources and runs slower than our proposal, despite the use of direct-mapped TLBs of lower capacity. The smallest of all the considered designs employs a software-managed TLB, but its performance is not among the best, as the TLB is searched sequentially [37]. Relying on a hardware-managed, virtually-addressed data cache to reduce the pressure on the low-capacity TLB and the memory [23] results in the highest memory consumption.

It has to be underlined that all the approaches that we compare to block memory traffic upon encountering the first (or second [32]) TLB miss, and until this miss has been handled. In contrast, our design delivers non-blocking operation. It simply enqueues the missing request, drops the transaction and continues to translate requests.

5.4.2 Alternative SVM Designs

The focus of previous work on SVM for FPGA accelerators lies on reducing the TLB service time by using either a soft processor [22], [24] or dedicated hardware [18], [19], [27], [28], [32], [37], [54] for managing the TLB with a size of 64 entries at most. As opposed to letting the host

7. The L2 TLB alone uses more than 60% of the resources.

TABLE 5
Comparison of different IOMMU FPGA implementations.

	LUTs* [k]	FFs [k]	BRAM [kbit]	#TLB Entries	Latency [Cycles]	Freq. [MHz]	Time [ns]	Latency [Host Cycles]	FPGA Family (Tech.)
This work	4.12	4.69	46.14	$4\,\&\,4+1024$	$1,\,3-6$	185	$5.4,\,16.2-32.4$	$3.6,\,10.8-21.6$	Kintex-7 (28 nm)
Estibals [27]	0.07 - 4.06	0.14 - 8.76	≤ 32	n/a	2	100	20.0	13.3	Artix-7 (28 nm)
Winterstein [54]	3.70	14.18	804.22	$64+64$	n/a	114	n/a	n/a	Cyclone-V (28 nm)
Shamani [39]	7.06	5.26	n/a	$8\,\&\,4+64$	2, 5	200	10.0, 25.0	n/a	Stratix-V (28 nm)
Mirian [37]	1.55	2.67	≤ 32	64	64	n/a	n/a	n/a	Virtex-6 (40 nm)
Kornaros [32]	10.62	407.65		64	6	225	26.7	26.7	Virtex-6 (40 nm)
Ammendola [24]	10.70	8.30	566.54	32	31	250	124.0	297.6	Stratix-IV (40 nm)
Ng [23]	2.74	2.84	1622.02	16	2	62	32.3	100.0	Virtex-5 (65 nm)

* Logic utilization on a Kintex-7 device. The scaling factors were obtained by synthesizing an FPGA accelerator design for the various device families.

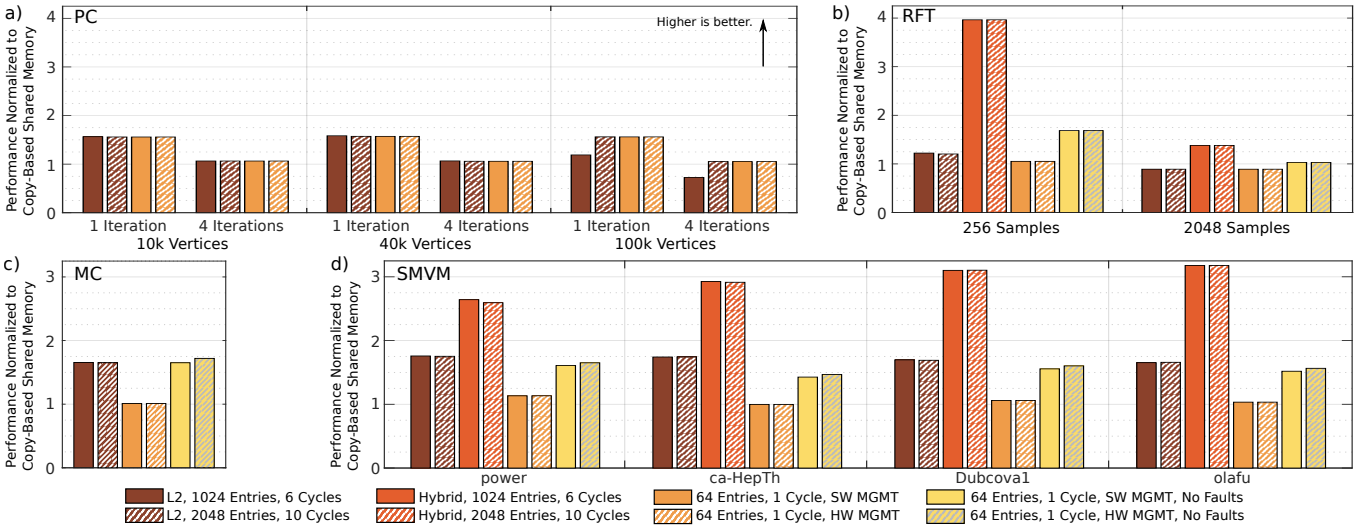

Fig. 11. Normalized performance of different SVM designs for a) PC, b) RFT, c) MC and d) SMVM.

manage the TLB, this reduces the latency of TLB misses substantially. However, the host is still required to pin the shared memory pages either on the first and compulsory TLB miss to a page (causing a page fault), or to all shared memory pages at offload time. Otherwise, the OS might at some point move the shared page in physical memory or to swap space, thereby causing the corresponding TLB entry to become invalid and causing the accelerator to corrupt the memory of other processes or the OS itself. In addition, most proposed designs support no outstanding TLB misses [23], [24], [27], [28], [37], [39], [54] – or just a single one [32] – without blocking any traffic. Compared to our design which simply enqueues outstanding TLB misses and in parallel continues to serve hitting memory transactions, this leads to a complete serialization of the traffic. Thus, the full interrupt latency has to be paid for every page fault.

To estimate the performance of such designs in real applications and compare them with our design, we have run our benchmarks on our Zynq-based evaluation platform using an L1 TLB with 64 entries and profiled the number of compulsory misses and capacity misses in the TLB. The total run time is then estimated by summing up the accelerator run time in the copy-based memory sharing configuration (no SVM-related overheads during accelerator execution) and the SVM-related overheads for the accelerator execution and the offloading sequence.

The cost of a page fault (compulsory miss, first miss to every page) is equal to the total TLB miss-handling cost in our scheme (Table 3). Further TLB misses (capacity misses) can be handled directly on the FPGA and take 1350 host clock cycles in the case of a soft processor managing the TLB [24] or 540 host clock cycles when the TLB is managed in hardware [28]. In case the shared memory pages can be pinned at offload time, there are no page faults. Note however that page pinning at offload time is only an option if the application does not make extensive use of pointer-rich data structures. Otherwise, an application-specific offload sequence is required to traverse the data structure (as with copy-based shared memory). The cost of pinning a single memory page at offload time is equal to the cost of handling a TLB miss in our scheme (Table 3, without response and scheduling latency) [20].

The accelerator engines have been configured as follows: For PC, we used a vertex size of 44 B, 10 cycles per vertex, and 1 to 4 iterations on the graph similar to PageRank [55]. We used 3 different graphs with 10 k, 40 k and 100 k vertices to vary the total data size. For RFT, we chose 16 tree levels, 28 B for the vertex size, 10 computation cycles per vertex and 256 or 2048 samples. MC was run with a data size of 64 KiB for 1 iteration to match the loading of the multi-channel image patches prior to an RFT-like classification phase [48]. For SMVM, we used the four matrices listed in Table 4.

PC: As shown in Fig. 11 a), the use of a soft processor (SW MGMT) or dedicated hardware (HW MGMT) together with a single-cycle, 64-entry TLB does not necessarily lead to higher performance. **The reason is mainly that compulsory TLB misses lead to page faults, which need to be handled by the host, even if the TLB is managed by the accelerator.** Independent of the SVM scheme, the speedup compared to copy-based shared memory decreases as the number of iterations on the graph increases which amortizes the initially high offload cost. Only as the graph size increases beyond the capacity of the large TLB (100 k vertices), managing the TLB on the FPGA starts to pay off. This is due to the much smaller latency for handling capacity misses directly on the FPGA. Selecting a larger TLB size (L2, 2048 Entries, 10 Cycles) in our system allows to avoid capacity TLB misses and the associated performance degradation. **The higher maximum look-up latency does not affect performance also for smaller graphs in PC.**

RFT: Since RFT does not make heavy use of virtual address pointers, the shared data can be pinned at offload time to avoid page faults. In this case, all TLB misses can be handled on the FPGA (64 Entries, 1 Cycle, SW/HW MGMT, No Faults) as shown in Fig. 11 b). The performance is close to that of copy-based shared memory also for larger numbers of samples. The highest performance is achieved by the hybrid design. A larger TLB with 10 instead of 6 cycles max look-up latency has little impact on performance (2%) only.

MC: MC features a linear access pattern to SVM and thus

only produces compulsory TLB misses and page faults but no capacity misses. **In our design, the IOMMU does not block on the first TLB miss. Together with the prefetching transactions, this allows to queue multiple TLB misses, which can be handled by the host in one batch.** This allows to increase performance by roughly 60% compared to when managing the TLB on the FPGA as shown in Fig. 11 c). Pinning the memory pages at offload time improves performance of such designs but it does so at most slightly beyond what is achievable with our framework.

SMVM: For SMVM, the performance is bound by handling compulsory misses/page faults as shown in Fig. 11 d). Pinning the shared data at offload time helps to improve performance. However, the performance of the designs managing the TLB on the FPGA is always below that of our design. The reason is twofold: First, these designs support neither queueing of page faults nor multiple outstanding TLB misses. While either of the two is being handled, the IOMMU blocks any traffic to SVM. **Second, SMVM uses multiple parallel DMA streams to stream in the sparse matrix in CISR format, which further amplifies the benefits of supporting multiple outstanding TLB misses.** The highest performance is achieved when using a hybrid design as enabled by our SVM framework.

6 Conclusion

In this paper, we have presented a plug-and-play framework for exploring shared virtual memory (SVM) for FPGA accelerators in heterogeneous embedded SoCs. It consists of a software component for interfacing the user-space application with the OS kernel and the address translation hardware. To perform the virtual-to-physical address translation of the accelerator's accesses to shared memory, it can use either our configurable IOMMU IP core or the hard-macro IOMMU provided by some next-gen high-end SoCs. The design allows the application programmer to simply share virtual address pointers with the FPGA accelerator without the need to use specialized memory allocators and the like. The low-level details are handled by the framework without incurring any offload-time overheads. We have evaluated the design using applications operating on pointer-rich data structures for which SVM is a *must* to enable reasonable design time effort, and that heavily stress the SVM subsystem.

The proposed design allows for speedups between 1.5 and 13x, compared to copy-based shared memory – which is still the state of the art for embedded, heterogeneous systems. Our results show that, due to limitations in the low-level drivers and kernel APIs, the performance of hard-macro IOMMUs can be dominated by handling page faults. Using our configurable soft IOMMU we have found that, unlike for CPUs, TLB look-up latency is not critical for FPGA accelerators – as the TLB is not in the accelerator's critical path to the scratchpad memory, but mainly used for latency-insensitive DMA transfers. Relaxing look-up latency allows for the construction of larger TLBs and thus lower miss rate and overall miss-handling overhead with less FPGA resources. Non-blocking support for multiple outstanding misses and batch-mode handling as well as flexible sized mappings are the capabilities of our design that proved the most beneficial for the performance of parallel FPGA accelerator architectures. The presented results further show that coherent access to the host's data caches is not always beneficial. While avoiding costly cache flushes, the effective bandwidth can be 45% lower compared accessing the data directly through a dedicated port on the DDR memory controller. For optimal performance and flexibility, our framework allows to optionally specify the interface to use on an address range or shared data element basis.

The presented framework is part of the open-source HERO platform (see https://pulp-platform.org).

Acknowledgments

This work was funded by the H2020 project HERCULES (No. 688860). The authors would like to thank Somaje Maheshwara Sharma and Conrad Burchert for the valuable work during their master and student projects.

References

[1] F. Saqib *et al.*, "Pipelined decision tree classification accelerator implementation in FPGA (DT-CAIF)," *IEEE Trans. Comput.*, vol. 64, no. 1, pp. 280–285, Jan 2015.
[2] H. Le and V. K. Prasanna, "A memory-efficient and modular approach for large-scale string pattern matching," *IEEE Trans. Comput.*, vol. 62, no. 5, pp. 844–857, May 2013.
[3] J. Stuecheli *et al.*, "CAPI: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7:1–7:7, 2015.
[4] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. 41st Int. Symp. Comput. Architecture*, 2014, pp. 13–24.
[5] B. Klauer, *The Convey Hybrid-Core Architecture*. New York, NY: Springer New York, 2013, pp. 431–451.
[6] H. Giefers and M. Platzner, "An FPGA-based reconfigurable mesh many-core," *IEEE Trans. Comput.*, vol. 63, no. 12, pp. 2919–2932, Dec 2014.
[7] H. Khdr *et al.*, "Power density-aware resource management for heterogeneous tiled multicores," *IEEE Trans. Comput.*, vol. 66, no. 3, pp. 488–501, March 2017.
[8] Xilinx Inc., "Zynq-7000 All Programmable SoC overview," Product Specification, 2016.
[9] Intel Corp., "Arria 10 Device Overview," Product Specification, 2016.
[10] A. Kurth *et al.*, "Mobile ultrasound imaging on heterogeneous multi-core platforms," in *Symp. Embedded Systems for Real-Time Multimedia*, Oct. 2016, pp. 9–18.
[11] P. Meloni *et al.*, "A high-efficiency runtime reconfigurable IP for CNN acceleration on a mid-range all-programmable SoC," in *Proc. Int. Conf. Reconfigurable Computing and FPGAs*, 2016, pp. 1–8.
[12] Xilinx Inc., "Zynq UltraScale+ MPSoC data sheet: Overview," Advance Product Specification, 2017.
[13] ARM Ltd., "ARM CoreLink MMU-500 system memory management unit," Technical reference manual, Cambridge, UK, 2016.
[14] Xilinx Inc., "SDSoC environment user guide," User Guide, 2017.
[15] AMD Inc., "2nd generation AMD Embedded R-Series APU," Product Brief, 2017.
[16] NVIDIA Corp., "NVIDIA Jetson TX2 delivers twice the intelligence to the edge," Parallel Forall blog, 2017, https://devblogs.nvidia.com/parallelforall/jetson-tx2-delivers-twice-intelligence-edge/.
[17] J. Vesely *et al.*, "Observations and opportunities in architecting shared virtual memory for heterogeneous systems," in *Proc. Int. Symp. Performance Analysis of Systems and Software*, 2016, pp. 161–171.
[18] K. Hsieh *et al.*, "Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation," in *Proc. 34th Int. Conf. Computer Design*, 2016, pp. 25–32.
[19] P. Mantovani *et al.*, "Handling large data sets for high-performance embedded applications in heterogeneous systems-on-chip," in *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, 2016, pp. 3:1–3:10.

[20] P. Vogel, A. Marongiu, and L. Benini, "Lightweight virtual memory support for many-core accelerators in heterogeneous embedded SoCs," in *Proc. 10th Int. Conf. Hardware/Software Codesign and System Synthesis*, 2015, pp. 45–54.
[21] P. Vogel, A. Marongiu, and L. Benini, "Lightweight virtual memory support for zero-copy sharing of pointer-rich data structures in heterogeneous embedded SoCs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 28, pp. 1947–1959, 2017.
[22] P. Vogel *et al.*, "Efficient virtual memory sharing via on-accelerator page table walking in heterogeneous embedded SoCs," *ACM Trans. Embedded Computing Systems*, vol. 16, no. 5s, pp. 154:1–154:19, 2017.
[23] H. C. Ng, Y. M. Choi, and H. K. H. So, "Direct virtual memory access from FPGA for high-productivity heterogeneous computing," in *Proc. Int. Conf. Field-Programmable Technology*, 2013, pp. 458–461.
[24] R. Ammendola *et al.*, "Virtual-to-physical address translation for an FPGA-based interconnect with host and GPU remote DMA capabilities," in *Proc. Int. Conf. Field-Programmable Technology*, 2013, pp. 58–65.
[25] Y. Choi *et al.*, "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms," in *Proc. 53rd Annual Design Automation Conf.*, 2016, pp. 109:1–109:6.
[26] IBM Corp., "Coherent accelerator processor interface user's manual," User's Manual, 2015.
[27] N. Estibals *et al.*, "System level synthesis for virtual memory enabled hardware threads," in *Proc. Design, Automation and Test Europe Conf. and Exhibition*, 2016, pp. 738–743.
[28] A. Agne, M. Platzner, and E. Lübbers, "Memory virtualization for multithreaded reconfigurable hardware," in *Proc. Int. Conf. Field-Programmable Logic and Appl.*, 2011, pp. 185–188.
[29] E. S. Chung, J. D. Davis, and J. Lee, "Linqits: Big data on little clients," in *Proc. 40th Int. Symp. Comput. Architeture*, 2013, pp. 261–272.
[30] M. Sadri *et al.*, "Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ," in *Proc. 10th FPGAworld Conf.*, 2013, pp. 5:1–5:8.
[31] S. Park, M. Kim, and H. Y. Yeom, "GCMA: Guaranteed contiguous memory allocator," *ACM SIGBED Review*, vol. 13, no. 1, pp. 29–34, Mar. 2016.
[32] G. Kornaros *et al.*, "I/O virtualization utilizing an efficient hardware system-level memory management unit," in *Proc. Int. Symp. System-on-Chip*, 2014, pp. 1–4.
[33] O. Peleg *et al.*, "Utilizing the IOMMU scalably," in *Proc. of USENIX Annual Technical Conf.*, 2015, pp. 549–562.
[34] J. Cong *et al.*, "Supporting address translation for accelerator-centric architectures," in *Proc. 23rd Int. Symp. High Perf. Computer Architecture*, 2017.
[35] Microsemi Corp., "SmartFusion2 SoC FPGA," Product Brief, 2017.
[36] M. Lavasani, H. Angepat, and D. Chiou, "An FPGA-based in-line accelerator for Memcached," *IEEE Comput. Archit. Lett.*, vol. 13, no. 2, pp. 57–60, 2014.
[37] V. Mirian and P. Chow, "Evaluating shared virtual memory in an OpenCL framework for embedded systems on FPGAs," in *Proc. Int. Conf. Reconfigurable Computing and FPGAs*, 2015, pp. 1–8.
[38] D. Nagle *et al.*, "Design tradeoffs for software-managed TLBs," *ACM SIGARCH Computer Architecture News*, vol. 21, no. 2, pp. 27–38, 1993.
[39] F. Shamani *et al.*, "Design, implementation and analysis of a run-time configurable memory management unit on FPGA," in *Proc. Nordic Circuits and Systems Conf.*, 2015, pp. 1–8.
[40] OpenMP Architecture Review Board. (2013) OpenMP application program interface. API specification.
[41] Khronos OpenCL Working Group. (2015) The OpenCL specification. Language specification.
[42] ARM Ltd., "AMBA AXI and ACE protocol specification," Protocol specification, 2013.
[43] M. Dehyadegari *et al.*, "Architecture support for tightly-coupled multi-core clusters with shared-memory HW accelerators," *IEEE Trans. Comput.*, vol. 64, no. 8, pp. 2132–2144, Aug 2015.
[44] Z. Ullah, "LH-CAM: Logic-based higher performance binary CAM architecture on FPGA," *IEEE Embedded Syst. Lett.*, vol. 9, no. 2, pp. 29–32, 2017.
[45] P. Yiannacouras, "An automatic cache generator for Stratix FPGAs," BASc Thesis, University of Toronto, 2003.
[46] J. Fowers *et al.*, "A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication," in *Proc. 22nd Annual Int. Symp. Field-Programmable Custom Computing Machines*. IEEE, 2014.
[47] Y. Guo *et al.*, "How well do graph-processing platforms perform? an empirical performance evaluation and analysis," in *Proc. 28th Int. Parallel and Distributed Processing Symp.*, 2014, pp. 395–404.
[48] J. Gall and V. Lempitsky, "Class-specific hough forests for object detection," in *Proc. Conf. Computer Vision and Pattern Recognition*, 2009, pp. 1022–1029.
[49] S. Schulter *et al.*, "On-line hough forests," in *Proc. 22nd British Machine Vision Conf.*, 2011, pp. 128.1–128.11.
[50] W. Yang *et al.*, "Performance optimization using partitioned SpMV on GPUs and multicore CPUs," *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2623–2636, Sept 2015.
[51] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E*, vol. 76, 2007.
[52] P. Erdős and A. Rényi, "On random graphs I." *Publicationes Mathematicae*, pp. 290–297, 1959.
[53] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Mathematical Software*, vol. 38, no. 1, pp. 1:1–1:25, 2011.
[54] F. Winterstein and G. Constantinides, "Pass a pointer: Exploring shared virtual memory abstractions in OpenCL tools for FPGAs," in *Proc. Int. Conf. Field-Programmable Technology*, 2017, pp. 1–8.
[55] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proc. 7th Int. World-Wide Web Conf.*, 1998, pp. 107–117.

Pirmin Vogel received his MSc degree in electrical engineering and information technology from ETH Zurich, Switzerland in 2013. Since then, he is pursuing his PhD degree at the Integrated Systems Laboratory (IIS), ETH Zurich. His research interests include digital signal processing, heterogeneous computing architectures and embedded systems-on-chip with a focus on operating system, driver, runtime and programming model support for efficient and transparent accelerator programming.

Andrea Marongiu received the MSc degree in electronic engineering from the University of Cagliari, Italy, in 2006 and the PhD degree in electronic engineering from the University of Bologna, Italy, in 2010. Since 2013 he has been a Research Fellow at ETH Zurich. He currently is an Assistant Professor at the University of Bologna. His research interests concern parallel programming model and architecture design in the single-chip multiprocessors domain, with special emphasis on compilation for heterogeneous architectures, efficient usage of on-chip memory hierarchies and SoC virtualization. He has published more than 80 papers in peer reviewed international journals and conferences.

Luca Benini holds the chair of digital Circuits and systems at ETHZ and is Full Professor at the Universita di Bologna. Dr. Benini's research interests are in energy-efficient system design for embedded and high-performance computing. He is also active in the area of energy-efficient smart sensors and ultra-low power VLSI design. He has published more than 800 papers, five books and several book chapters. He is a Fellow of the IEEE and the ACM and a member of the Academia Europaea. He is the recipient of the 2016 IEEE CAS Mac Van Valkenburg award.