Efficient virtual memory sharing via on-accelerator page table walking in heterogeneous embedded SoCs

Efficient Virtual Memory Sharing via On-Accelerator Page Table Walking in Heterogeneous Embedded SoCs

PIRMIN VOGEL, ANDREAS KURTH, and JOHANNES WEINBUCH,
Integrated Systems Laboratory, ETH Zurich
ANDREA MARONGIU and LUCA BENINI,
Integrated Systems Laboratory, ETH Zurich and
Electrical, Electronic, and Information Engineering Department, University of Bologna

Shared virtual memory is key in heterogeneous systems on chip (SoCs) that combine a general-purpose host processor with a many-core accelerator, both for programmability and performance. In contrast to the full-blown, hardware-only solutions predominant in modern high-end systems, lightweight hardware-software co-designs are better suited in the context of more power- and area-constrained embedded systems and provide additional benefits in terms of flexibility and predictability. As a downside, the latter solutions require the host to handle in software synchronization in case of page misses as well as miss handling. This may incur considerable run-time overheads.

In this work, we present a novel hardware-software virtual memory management approach for many-core accelerators in heterogeneous embedded SoCs. It exploits an *accelerator-side* helper thread concept that enables the accelerator to manage its virtual memory hardware autonomously while operating cache-coherently on the page tables of the user-space processes of the host. This greatly reduces overhead with respect to host-side solutions while retaining flexibility. We have validated the design with a set of parameterizable benchmarks and real-world applications covering various application domains. For purely memory-bound kernels, the accelerator performance improves by a factor of 3.8 compared with host-based management and lies within 50% of a lower-bound ideal memory management unit.

CCS Concepts: • **Software and its engineering** → **Virtual memory**; *Main memory*; • **Computer systems organization** → *Heterogeneous (hybrid) systems*; *System on a chip*; *Embedded software*;

Additional Key Words and Phrases: Shared Virtual Memory, TLB Management, Linux, Heterogeneous SoCs, Embedded Systems

ACM Reference format:
Pirmin Vogel, Andreas Kurth, Johannes Weinbuch, Andrea Marongiu, and Luca Benini. 2017. Efficient Virtual Memory Sharing via On-Accelerator Page Table Walking in Heterogeneous Embedded SoCs. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (October 2017), 19 pages.

1 INTRODUCTION

Heterogeneous embedded systems on chip (HESoCs) are increasingly being adopted in various application domains due to their high energy-efficiency and peak performance. Such architectures

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue.
This work was funded by the H2020 project HERCULES (No. 688860).

DOI: 0000001.0000001

combine a general-purpose host processor with a massively parallel programmable many-core accelerator (PMCA) [1, 16, 22] and are nominally capable of achieving extremely high GOPS/Watt.

Historically, the complex memory systems adopted by HESoCs has brought the biggest difficulties in application development. On the host side, coherent caches and memory management units (MMUs) make the memory hierarchy completely transparent. On the PMCA side, however, scratchpad memories (SPMs) are physically addressed and need to be explicitly managed via direct memory access (DMA) transfers. When data is shared between the two sides, data consistency also requires explicit coherency operations (e.g., host cache flushes).

If partitioning program data for DMA transfers represents a significant effort for applications with regular memory access patterns, it literally becomes a nightmare for irregular programs based on complex data structures (e.g., pointer chasing). Offloading pointer-intensive computations to PMCAs is incredibly burdensome, as data structures spanning several virtual memory pages must be moved to contiguous, unpaged, and uncached memory areas. On top of that, any virtual address stored inside the data structures (typically initialized on the host side) needs to be adjusted to point to the copy in physically contiguous memory. Practically, this requires traversing the entire data structure at run time, which not only hampers programmability but also kills performance [7].

In the past few years, initiatives such as the Heterogeneous System Architecture (HSA) Foundation [12] have been pushing for an architectural model where the host processor and the accelerator(s) communicate via coherent shared virtual memory (SVM). Nowadays, virtually all major embedded GPU vendors are equipping their SoCs with full-fledged HW support for SVM, consisting of input/output MMUs (IOMMUs) and coherent interconnections. In the low-end embedded SoC domain, however, SVM is not or only partially supported. Specifically, SVM solutions for HESoCs usually rely on mixed hardware-software designs, where simple input/output translation lookaside buffers (IOTLBs) used by the accelerator to translate virtual addresses are fully controlled by software (e.g., via a kernel-level driver module) [19, 31].

Compared with full-featured IOMMUs, these solutions are better suited in the context of constrained HESoCs as they largely reduce area and energy overheads. Moreoever, they offer much greater flexibility than possible with the reactive miss handling performed by hardware IOMMUs: software-managed virtual memory can potentially exploit higher-level information of the application to anticipate the access pattern to shared memory and thus hide latency and increase predictability of the system. However, their simple design comes at the price of considerable runtime overheads for some workloads. One of the key sources of overhead for state-of-the-art HESoC SVM support is remote miss handling [31]. Here, if a processing element (PE) in the PMCA creates a miss, this particular PE is put to sleep and the host is notified to handle the miss. *The inability to handle misses locally on the accelerator is the main limitation of such solutions, causing delays of thousands of clock cycles due to interrupt-based accelerator-to-host communication.*

In this work, we present an integrated HW/SW solution that enables on-PMCA page table walking to efficiently implement IOTLB miss handling on HESoCs. Operating on the host's page table of the offloaded user-space process, the design allows the accelerator to autonomously manage the shared translation lookaside buffers (TLBs) without host intervention. The design is flexible and offers the possibility for collaborative TLB management, e.g., to exploit application-level knowledge available at offload time on the host side. Our solution is based on a software implementation of a miss handling thread (MHT), which relies on an underlying virtual memory management library that encapsulates all the functionality for tightly-coupled interaction with the HW, and requires only minimal hardware modifications to the Remapping Address Block (RAB) proposed in [31].

We have validated our design with a set of parameterizable synthetic benchmarks and real-world applications capturing the most relevant memory access patterns from regular and irregular parallel

workloads. Compared with host-based TLB management, the accelerator performance improves substantially. Even for purely memory-bound kernels, the performance lies within 50% of an ideal design, i.e., a TLB with zero look-up and miss latency, which is out of reach even for today's hardware IOMMUs [8].

The rest of the paper is organized as follows. § 2 discusses related work. The target architecture template is described in § 3, and § 4 gives the necessary background information. The main problem of state-of-the-art solutions is stated in § 5. We present our solution to this problem in § 6. § 7 gives experimental results, and § 8 concludes the paper.

2 RELATED WORK

The industry's approach for SVM is that of full-blown, hardware-only solutions targeting maximum performance in high-end SoCs [2, 5, 14]. On the other end of the spectrum, i.e., in low-power, embedded SoCs, heterogeneous SVM is not yet widely adopted. At the time of writing, the Xilinx Zynq UltraScale+ MPSoC is becoming available in the form of engineering samples [34]. While this next-generation flagship HESoC offers a full-fledged hardware IOMMU [5], it currently comes with limited software and tool support. The tool documentation [33] does not mention the possibility to use this IOMMU for giving a loosely-coupled accelerator direct access to user-space memory, which would further require additional support in the form of drivers and libraries that is currently missing. Instead, the host is meant to initiate the data movement from and to the SPMs of the accelerator through the Linux DMA application programming interface (API) [33]. The IOMMU is set up on a DMA transfer basis from within the DMA API, which involves the generation of a dedicated I/O page table. As such, the IOMMU continues to serve its original purposes of isolating the host system from malicious or faulty DMA devices and drivers [25] and of providing the DMA engine with the illusion of a physically contiguous memory for improved performance. This is not sufficient to enable efficient data sharing between host and accelerator.

To enable memory sharing, today's embbeded systems stick to simpler solutions which do not offer the convenience of SVM to the programmer and are often not zero-copy. An obvious way for sharing memory in a heterogeneous system is to allocate a large, physically contiguous memory section for the data to be shared. Virtual-to-physical address translation then simplifies to applying a constant offset. With the Contiguous Memory Allocator (CMA), Linux has such a mechanism on board [23]. A kernel-level driver can request memory from a pre-allocated physically contiguous memory section and give access to it to user-space applications through, e.g., an *mmap()* system call. While this allows for zero-copy data sharing, it suffers from limited performance: The kernel may need to first move other data out of the pre-allocated section before the contiguous section can be made available to the accelerator, which can incur a long latency [24]. Also, there is no guarantee that a pre-allocated memory region can be made available at all [9]. Finally, the CMA returns uncached memory on some widely adopted processor architectures like ARMv7. Letting the host do computations on the shared buffer is thus very inefficient and should be avoided.

In practice, heterogeneous embedded systems often stick to copy-based, physically shared memory where the main memory is split into two sections [18]: one exclusively accessed by the host and a second, physically contiguous, uncached section accessible also by the accelerator. Before and after offloading computation to the accelerator, the host must copy the shared data between the two sections and modify any virtual address pointers inside the shared data, which is cumbersome for the application developer and slow in performance.

There are different approaches in the embedded systems domain to enable true SVM. These include full-fledged hardware IOMMUs [5, 17] equipped with coherent translation caches and large buffer memories to absorb memory transactions missing in the TLB. While this approach allows

to completely abstract away the underlying SVM system from the accelerator, its scalability to embedded many-core accelerators featuring many independent DMA engines and its applicability to more resource-constrained systems is questionable. Alternative mixed hardware-software designs rely on placing the shared data in contiguous memory using a specific user-space API [13] or by replacing the standard *malloc()* with a customized implementation [21]. At offload time, a kernel-level driver on the host then generates an optimized page table or DMA transaction descriptors for the accelerator. In constrast, more lightweight and less intrusive solutions suggest the use of a hardware IOTLB that is completely software-managed by a kernel-level driver on the host [19, 31]. As a downside, the frequent host interaction and the associated interrupt latency impose substantial accelerator run-time overheads.

To overcome the inefficiencies of the IOMMUs found in today's processors, a new IOMMU architecture for accelerator-centric systems has been proposed that uses local TLBs but offloads the page table walks to the MMUs of the CPU requiring modifications to the host hardware [8]. Studies on SVM solutions for general purpose GPUs (GPGPUs) [26, 27] and FPGA-based accelerators [29] led to IOMMU designs targeting high-end systems and featuring (multi-threaded) hardware page table walk engines as well as prefetching logic and coherent caches. This allows to reduce the host interaction to a minimum required for maximum performance.

This work reduces the performance gap between hardware IOMMUs and software-managed SVM designs targeting HESoCs. It neither requires modifications to the host architecture nor restricts the programmer to use dedicated memory allocators. Low-level details are taken care of by the infrastructure without additional overheads at offload time.

3 TARGET ARCHITECTURE TEMPLATE

The heterogeneous embedded system targeted in this work combines two architecturally different processing units on a single chip. As shown in Fig. 1, the central component of this HESoC is a powerful general-purpose multi-core CPU (the *host*), which is equipped with a multi-level cache-coherent memory hierarchy and runs a full-fledged operating system (OS). To improve overall performance/Watt, the HESoC can offload critical computation kernels to a PMCA that consists of several tens of simple PEs [1, 22].

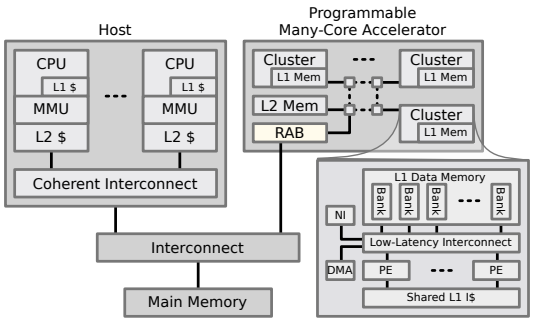

Fig. 1. High-level view on the target architecture.

To allow for a highly scalable architecture, the PMCA we consider uses a multi-cluster design [16, 22]. Per cluster, multiple PEs share an L1 instruction cache and an L1 data SPM, both multi banked. The shared L2 SPM as well as the L1 SPMs of the other clusters can be accessed through the global interconnect, albeit at a higher latency. The main, off-chip dynamic random access memory (DRAM) is physically shared among the host and the PMCA [12]. To constantly feed the various processing

engines with data and exploit the available computing resources, both the host and the PMCA use their internal memory hierarchy to keep the most frequently accessed data in fast, local storage. The host does so using its hardware-managed cache hierarchy. In contrast, the PMCA relies on multi-channel, high-bandwidth DMA engines and double-buffering schemes to overlap data movement with actual computation performed on data available in the cluster-internal L1 SPM.

The accelerator uses a Remapping Address Block (RAB) [31] for interfacing the shared main memory which allows the host and the accelerator to exchange virtual address pointers. As opposed to full-fledged hardware IOMMUs [5, 26, 27], the RAB is essentially an IOTLB that is completely managed by software.

4 BACKGROUND

In this section, we discuss the specifics and challenges of managing virtual memory (VM) on a PMCA in a HESoC on the basis of a system with a 32-bit ARM host processor running Linux. We first discuss structure and entries of the page table (PT) and how the PT is used for virtual-to-physical address translation. As the PT can be cached at different locations, we then discuss the related coherence issues. Next, we discuss the RAB, a lightweight hardware module that enables VM on PMCAs. We conclude this section by examining the limitations of the state-of-the-art solution and derive the challenges our implementation (in § 6) has to solve.

4.1 Linux Page Table

PTs store the virtual-to-physical address translations of all processes. There is one PT per process. All PTs are managed by the OS, which creates, modifies, and removes PT entries. In addition to the OS, the PT is read by the MMU of the CPU. The layout of the PT is defined by the CPU architecture. The Linux kernel generally uses the four-level PT shown in Fig. 2. The individual tables are called Page Global Directory (PGD), Page Upper Directory (PUD), Page Middle Directory (PMD), and Page Table Entry (PTE).

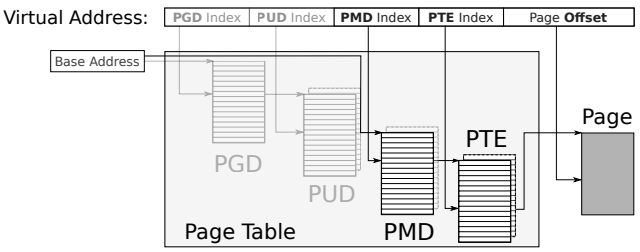

Fig. 2. The Linux page table. Elements that are grayed out are unused the host architecture used in this paper.

The host processor we use, a 32-bit ARMv7 CPU without Large Physical Address Extension (LPAE), however, uses a two-level PT. To match this architecture, Linux uses only the lower two of its four tables (i.e., the PMD and the PTE) and sets the base address of the PT to the base address of the PMD. Another difference between processor and kernel is the number of entries per table: The Linux kernel uses 2048 and 512 entries in the first and the second table, respectively. The processor, on the other hand, uses 4096 and 256 entries in the two levels. The kernel compensates for this difference by duplicating the information in the second table and by defining additional entries suitable for the processor in the first table. We will not discuss this mechanism in more detail because we work with the Linux representation of the PT. As a consequence, our implementation can be adapted for all host architectures that are supported by the Linux kernel.

Entries in the upper-level table contain the physical base address of a lower-level table. Entries in the lower-level table contain the physical base address of a page and multiple flag bits, of which the read-only and the user-mode bit are relevant for us: If the user-mode bit is zero, unprivileged processes (e.g., all user-space applications) may not access the page. If the read-only bit is set, processes may not write the page. If the physical address of an entry is null, the given virtual address (VA) does not translate to a physical address.

The translation from a virtual to a physical address for a given process is done in a page table walk (PTW): First, the 11 most-significant bits of the VA are used to index the PMD that is located at the process-specific PT base address to get the physical base address of the PTE. Second, the next 9 bits of the VA are used to index the PTE to get the physical base address of the page. Finally, the 12 least-significant bits of the VA are added to the page base address to obtain the physical address.

4.2 Page Table Coherence

The PTs of all processes are stored in main memory due to their size. However, the virtual-to-physical address translations they contain are used very often, and accessing main memory on every translation would cause huge run-time overheads. To avoid this, the most frequently used translations are cached.

There are two types of caches: (IO)TLBs store individual address translations and are a core component of every (IO)MMU. They allow the direct translation of a set of recently used addresses without having to walk the PT. In addition to TLBs, parts of tables may be present in caches in the memory hierarchy after a PTW. Similarly, some high-performance (IO)MMUs even include private caches to buffer entire tables [5]. These caches speed up (a part of) the PTW, because some tables are available in local low-latency memory. Regardless of their type, caches introduce the problem of coherence: when the OS modifies an entry in the PT in main memory, all cached versions of this entry are invalid and may no longer be used.

To ensure this, an invalidation is broadcast whenever a PT entry is changed [20]. Caches are obliged to listen to these broadcasts and may not use invalidated entries any longer. There is a subtle difference in how the two cache types deal with invalidations: Caches in the memory hierarchy are subject to a cache-coherency protocol. TLBs, on the other hand, are not part of the memory hierarchy and thus cannot use cache invalidation broadcasts. To invalidate a TLB entry, an additional TLB invalidation that contains the modified virtual address is broadcast. On the ARMv7 host processor used in this paper, TLB invalidations are propagated through distributed virtual memory (DVM) transactions [4].

4.3 Remapping Address Block

The Remapping Address Block (RAB) [31] connects the PMCA to the shared main memory interconnect (see Fig. 1). Similar to the IOMMUs found in high-performance systems, the RAB performs the virtual-to-physical address translation for the accesses of the accelerator to the shared main memory. Unlike a full-fledged hardware IOMMU, however, the RAB is completely managed in software by a kernel-level driver module running on the host.

The top-level architecture of the RAB is shown in Fig. 5 (uncolored elements). At its heart is an IOTLB of configurable size. This IOTLB is fully associative and looks up translations in a single clock cycle. An IOTLB entry (called *RAB slice*) contains a virtual address range of arbitrary length, a physical address offset, and two flag bits: slice and write enable. The widely-adopted AXI4 [4] protocol is used on all interfaces.

When a transaction from the accelerator to shared memory arrives on the slave port, the metadata of the transaction (i.e., virtual address, length, ID, and read/write) are fed to the control block that

interfaces the IOTLB. In case of a RAB miss—i.e., if the virtual address does not match any enabled RAB slice or if a write has been requested but is not permitted—the RAB drops the transaction, responds with a slave error, stores the metadata inside the miss FIFO queue, raises a miss interrupt, and processes the next transaction. In case of a RAB hit, the RAB translates the virtual address of the transaction to the physical address, forwards the transaction through the master port, and processes the next transaction (returning the response of the downstream slave as it becomes available).

Supporting RAB misses requires a non-intrusive extension of the cluster: The RAB response is buffered in a low-latency, PE-private *TRYX* register. Each PE reads this register after every virtual memory access to check for a RAB miss. In case of a miss, the corresponding PE goes to sleep. It is woken up as soon as the miss is handled and then retries the memory access. The necessary instructions are automatically inserted by a compiler extension. Other PEs and DMA engines can continue accessing SVM while the miss is being handled.

In the state-of-the-art implementation [31], RAB misses are handled by the host CPU. For this, the miss interrupt and the configuration interface of the RAB are connected to the host, and a kernel driver is registered to handle that interrupt. When an interrupt occurs, the driver schedules the miss handler routine and clears the interrupt. When invoked, the handler then reads the miss FIFO in the RAB, walks the PT of the offloaded process, sets up a new RAB slice, and wakes up the PE that has caused the miss.

5 THE PROBLEM: HOST-BASED RAB MISS HANDLING

While handling RAB misses on the host comes at minimal hardware costs and is simple from the view of the PMCA, it adds an extensive run time overhead to every miss. On our evaluation platform, handling a single RAB miss on the host takes an average of 5400 PMCA clock cycles. Fig. 3 shows the interactions (top) and the timeline (bottom) of host-based miss handling.

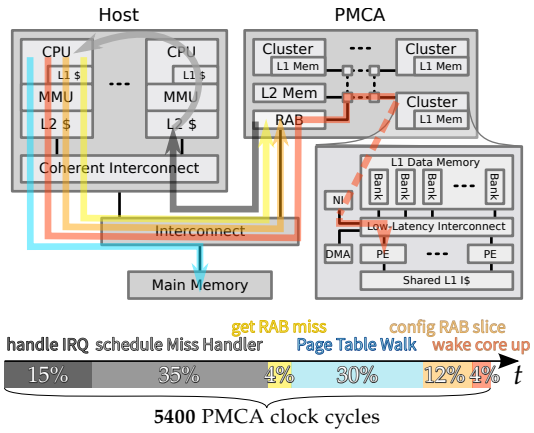

Fig. 3. Interactions (top) and timeline (bottom) of handling RAB misses on the host.

After a RAB miss, an interrupt from the RAB arrives at the host and is handled by one of the CPUs (dark gray); this takes ca. 15% of the total time. Then, the kernel thread executing the RAB miss handler in the process context is scheduled (light gray), which takes another 35% of the time. This step is required because some functions used in the miss-handling routine may sleep and thus cannot be executed in interrupt context. The miss handling routine first reads the RAB

miss FIFO (yellow, ca. 4% of the time) and then translates the virtual to a physical address using *get_user_pages()* (blue, ca. 30% of the time). Next, the routine configures a RAB slice in a series of transactions (orange, ca. 12% of the time) and finally wakes the sleeping PE up (red, ca. 4% of the time). In conclusion, there are two main contributors to the prohibitively large run time overhead: multiple interactions between loosely-coupled components and the long scheduling delay to execute the miss handler on the host.

Our solution, described in the next section, reduces the extensive run time overhead of the state-of-the-art design while maintaining the standard compliance and flexibility of the host-based solution. This will make VM viable also for applications that use VM extensively and potentially cause many RAB misses.

6 HW/SW SOLUTION FOR HANDLING RAB MISSES ON THE ACCELERATOR

The PMCA uses a multiple instruction multiple data (MIMD) execution model as visualized in Fig. 4. Multiple parallel worker threads mainly operate on data in the shared SPM. Accesses to SVM are protected by load and store macros (1) automatically inserted by a compiler extension [31]. These macros protect a PE from using an invalid response returned by the hardware upon a RAB miss by putting the PE to sleep. To reduce the problematic overhead caused by handling RAB misses on the host, we must instead handle misses right where they occur: on the PMCA. For this, we add a miss handling thread (MHT) to the execution model. Upon a RAB miss (2), the runtime starts an MHT (3) that handles all outstanding RAB misses through our virtual memory management (VMM) library. After every handled miss, the MHT sends a signal to the runtime that wakes up the sleeping PE (4). In addition, worker threads can explicitly call the VMM library to map and unmap pages, e.g., before setting up DMA transfers (5).

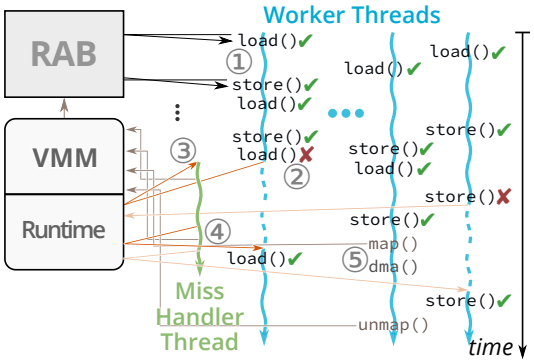

Fig. 4. Execution model on the PMCA with a miss handling thread (MHT).

To let the PMCA operate its VM hardware autonomously through the VMM software library (§ 6.3), two small modifications to the RAB hardware as proposed in [31] are required. First, the PMCA must be able to read the PT of the user-space process in main memory coherently with the caches of the host (§ 6.1). Second, the PMCA must have access to the configuration interface of the RAB (§ 6.2).

Finally, handling RAB misses on the PMCA instead of on the host requires two small changes to the driver on the host: During the offload, the driver must set up a RAB slice that enables the PMCA to access the PT. For this, the driver gets the physical address of the initial level of the PT (i.e., in our case, the PMD) and sets up a cache-coherent read-only slice for it. Additionally, the RAB miss interrupt handler on the host must be deactivated.

6.1 Page Table Coherence

There are two issues related to page table coherence (§ 4.2): all accesses to the PT must be coherent with the caches of the host, and all TLBs must respect TLB invalidations. We solved these issues as follows:

6.1.1 Cache-Coherent Memory Accesses. The PMCA accesses the PT coherently with the caches of the host for two reasons: First, this removes the need to flush caches every time the host changes the PT while the accelerator is running and on every offload. Second, this allows to exploit the memory hierarchy of the host to cache parts of the PT, reducing the average access time to the PT without the expensive dedicated hardware caches as used, e.g., in some IOMMUs.

Many modern SoCs offer an Accelerator Coherency Port (ACP) that enables accelerators to access main memory coherently with the caches of the host [15, 32]. To access the PT coherently, we could route all memory accesses through the RAB to the ACP. However, application data may be known to be uncached by the host at accelerator run time. In this case, accessing memory through the host's caches unnecessarily increases the access latency and causes interference. Thus, we added a multiplexer that routes cache-coherent accesses through the ACP and non-coherent accesses directly to the DDR memory interface (drawn yellow in Fig. 5). To control this multiplexer, we added one bit that defines cache coherence to each RAB slice.

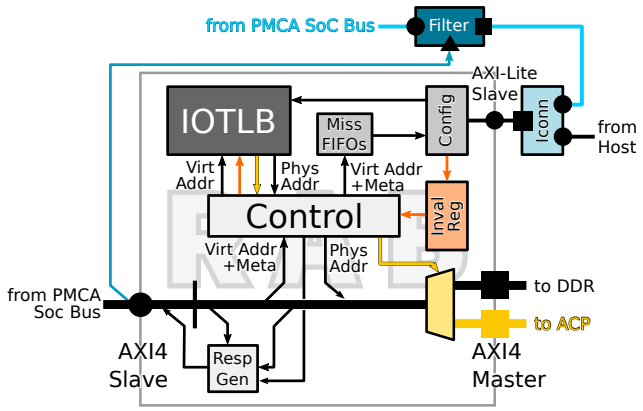

Fig. 5. The RAB with our hardware extensions highlighted in color.

6.1.2 TLB Invalidations. Ensuring coherence of the IOTLB in the RAB is more involved, because the RAB does not support TLB invalidations through DVM transactions. We solved this problem with another low-overhead hardware-software solution: We added a register to the RAB (drawn orange in Fig. 5) and amended the TLB invalidation routine in the Linux kernel to write the invalidated virtual address to that register. When an address is entered in the register, the RAB disables the corresponding slice within the next clock cycle and before processing any outstanding translations. A single register is sufficient because the RAB processes invalidations at a much higher rate than the OS generates them. This solution requires only little extra hardware and, since TLB invalidations happen rarely, the associated run time overhead is acceptable: On the evaluation platform (described in § 7.1), writing the invalidation register in the PMCA takes 200 clock cycles on average. This is comparable to the invalidation of a host TLB entry that takes 130 cycles on average.

6.2 Securely Configuring the RAB on the PMCA

The PMCA requires hardware access to the RAB configuration to be able to set up slices. To share the configuration port of the RAB among host and PMCA, we added an interconnect in front of the port and attached the PMCA SoC bus and the existing host master (light blue in Fig. 5). While this solution is simple and functionally sufficient, it would elude the isolation aspect of VM: If the PMCA was free to write any configuration to RAB slices, any application running on the PMCA could grant itself unrestricted access to any memory region, including regions that belong to the OS kernel.

We prevent illegal RAB configurations with a hardware filter on the configuration port (dark blue in Fig. 5). This filter ensures two properties based on address and content of incoming writes: First, only the host is allowed to write the one slice that maps the initial level of the PT. Second, the accelerator may only configure slices according to the response of a PT lookup. To ensure the second point, the filter snoops the read request and response channels between the RAB and the clusters of the PMCA (triangular filter port in Fig. 5) for lookups in the PT. The physical address, the access permissions, and the address range returned by a lookup are stored. On the next write to the RAB configuration by the accelerator, the values to be written are compared with the stored values of the last lookup: If one of them does not match or if the page is not accessible from user space, the write request is dropped and an error is returned. Applications could still escape this protection by faking an extra level in the PTW with one of their data pages. To prevent this, the filter keeps track of the number of indirections that occur during the PTW, which is fixed for any given host architecture.

This lightweight hardware solution enables untrusted accelerator software to configure the IOTLB while maintaining the isolation guarantee of VM. It does not have any impact on run time and comes at negligible hardware costs.

6.3 VMM Library for the PMCA

There are three basic requirements on this library that is located between the application programmer and PMCA hardware and host-dependent data structures. First, the library must expose a simple API that remains stable regardless of variations between different host architectures or evolution in PMCA hardware. Second, the implementation of the library must be flexible and extensible to accommodate different host architectures and evolving PMCA hardware. Third, the library is highly performance-critical and must guarantee correctness properties such as strict ordering of memory transactions. To meet the first two requirements, the library (shown in Fig. 6) is organized in layers and composed of modules. To meet the third requirement, we implement these abstractions with no overhead: the compiler expands all function calls inline and reduces all data structures to collections of primitive types. In the remainder of this section, we will discuss the modules and their interaction from top to bottom.

6.3.1 Top-Level VMM Module. The VMM library provides two different sets of functionalities in its interface: one to handle all outstanding RAB misses and automatically map the required pages (V1) and one to manually map and unmap pages (V2). In the most common use case of this library, an MHT calls the function to handle RAB misses (V1) upon a RAB miss interrupt. The library will then dequeue the first miss from the RAB miss queue and translate its virtual address to a physical address. Next, it will select a RAB slice to replace and configure that slice accordingly. Finally, it will wake up the PE that caused the miss. The other use case is the explicit mapping of pages (V2), e.g., in preparation of a DMA transfer. This works the same as (V1) without the functions related to miss handling.

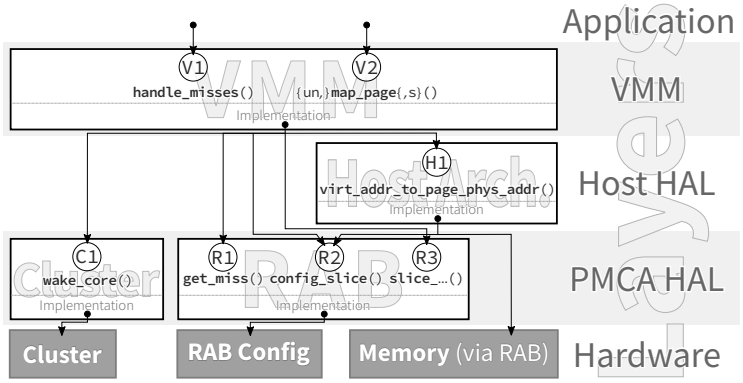

Fig. 6. Overview of the VMM library on the PMCA. Arrows indicate usage of functions and data structures in the interface.

An advantage of this software implementation is that the algorithm to replace slices can be modified easily. We focused on a simple FIFO policy due to its low time and space complexity, but certain applications could benefit from a more complex algorithm such as least recently used (LRU).

6.3.2 Host Hardware Abstraction Layer. The host hardware abstraction layer (HAL) provides a normalized interface to translate addresses on different host architectures. Its implementation structure allows to add support for every architecture supported by the Linux kernel with little extra development efforts.

This module translates virtual to physical addresses (H1) by walking the PT of the host. The translation starts at the first-level table, which has been mapped by the host at offload. At each level, the function first calculates the table index from the virtual address and reads the corresponding table entry. It then checks if this entry contains the physical address of a table in the next-lower level; if so, it sets up a new RAB slice for cache-coherent read-only access to that table and proceeds with the next level. At the last level, the table entry contains the physical address of a page along with access permissions. Thus, the module finally checks if the page is accessible from user space and maps it with the appropriate read/write permissions.

At all levels but the last, a RAB slice mapping a table is set up. When the translation of a virtual address (at least partially) requires the same tables as the previous address, however, setting up these tables again is unnecessary. Our implementation avoids such redundancy by caching the addresses of the currently mapped tables and starting the PTW at the first level that cannot be reused. This is especially beneficial for higher-level tables, which are reused often because they cover a large range of addresses. Our results show that this optimization already for a two-level PT reduces the run time of a PTW by up to 65% on the evaluation platform.

6.3.3 PMCA Hardware Abstraction Layer. The PMCA HAL implement the uniform access principle for memory-mapped hardware components of the PMCA by providing a consistent interface. This allows aggressive compiler optimization throughout this performance-critical library while still guaranteeing correctness at the instruction level, e.g., with respect to memory access ordering: constructs such as volatile variables and fixed-width types are used consistently where required—and only where required.

The RAB module controls RAB slices (R2) and dequeues misses from the hardware FIFO (R1). Parameters for a RAB slice are: the index of the slice, the virtual address range, the base physical

address, access permissions, and cache coherency. If the parameters are valid, the implementation then disables the target slice, writes the new configuration in a sequence of memory transactions, and re-enables the slice. Other operations on slices are (R3): disabling it, determining whether it maps a given virtual address, and determining whether it is enabled. The implementation is thread-safe: those operations that require more than one transaction lock the slice on which they operate by means of synchronization hardware in the PMCA.

From the cluster module, the VMM library currently only uses the function to wake up a given PE (C1) that has gone to sleep after accessing a virtual address that misses in the RAB. This function triggers an interrupt to the PE in the event hardware unit of the PMCA.

7 RESULTS

Before discussing the results, this section gives an overview of the evaluation platform and the benchmark applications visualized in Fig. 7 and 8, respectively.

7.1 Evaluation Platform

Our evaluation platform is based on the Xilinx Zynq-7000 SoC [32]. It features a dual-core ARM Cortex-A9 CPU that is used to implement the *host* of the HESoC. The cores have separate L1 instruction and data caches with a size of 32 KiB each. Further, they share 512 KiB of unified L2 instruction and data cache including a controller that connects to the high-priority port of the DRAM controller. The host is running Xilinx Linux 3.18 with swapping and the LPAE disabled.

The programmable logic of the Zynq is used to implement a cluster-based PMCA architecture [28]. The PEs within a cluster feature 8 KiB of shared L1 instruction cache and share 256 KiB multi-banked tightly-coupled data memory as L1 SPM. Ideally, every PE can access one word in the L1 SPM per cycle. Every cluster features a multi-channel DMA engine that can be quickly programmed by the PEs by just writing 4 low-latency, PE-private registers and thus enables fast and flexible movement of data between L1 and L2 memory or shared DRAM.

The event unit in the peripherals of each cluster is used for intra- and inter-cluster synchronization, to put PEs to sleep, and to wake them up. The PMCA is attached to the host as a memory-mapped device and is controlled by a kernel-level driver module and a user-space runtime. The host and the PMCA share 1 GiB of DDR3 DRAM. The RAB uses one port with 32 slices for PMCA-to-host communication. It connects the PMCA to the high-performance AXI slave port of the DRAM controller and to the ACP of the Zynq. The latter allows the PMCA to access the shared main memory coherent to the data caches of the host. The configuration interface of the RAB is accessible to both PMCA and host through the general-purpose AXI master port. In case a RAB miss happens, this is signaled back to the PE using the per-core, low-latency *TRYX* register.

This platform enables us to study and evaluate the system-level integration of a PMCA into a HESoC. Thus, we did not optimize the PMCA for the implementation on the FPGA; the FPGA implementation should be seen as an emulator instead of a full-featured accelerator. We adjusted the clock frequencies of the different components to obtain ratios similar to a real HESoC with host and PMCA running at 2133 MHz and 500 MHz, respectively. The DDR3 DRAM is clocked at 533 MHz. The hardware modifications to the RAB [31] add only minor contributions to the overall hardware complexity. The modified RAB accounts for less than 6% and 10% of the look-up table and flip-flop resources of the entire PMCA featuring a single cluster with eight PEs, respectively.

For future evolutions of our SVM system, we are currently migrating to a bigger evaluation platform combining a more recent, multi-cluster, ARMv8 host processor with an FPGA fabric capable of implementing multiple PMCA clusters. Since the VMM library presented in this work operates on the Linux PT, it is fully compliant with the new host architecture.

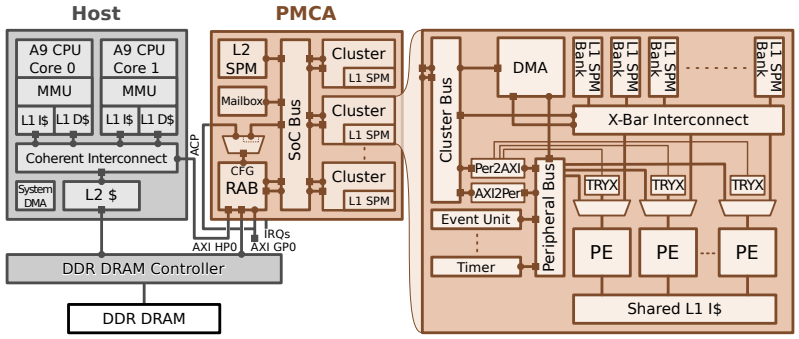

Fig. 7. Architecture of the evaluation platform.

7.2 Benchmark Description

To evaluate the performance of our SVM system under various conditions including identifying its limits, we have used three configurable benchmark applications. They were obtained by extracting critical phases from real-world applications suitable for the implementation on a HESoC, and by parameterizing them to cover a large parameter space. They exhibit main-memory access patterns representative for various application domains. Operating on pointer-rich data structures, the support for SVM is fundamental to allow implementations on heterogeneous platforms at a reasonable effort. Otherwise, developers need to completely rethink the application, which is a huge pain. Moreover, the access patterns exhibited by such operations are highly irregular and thus represent a worst-case scenario for the SVM subsystem.

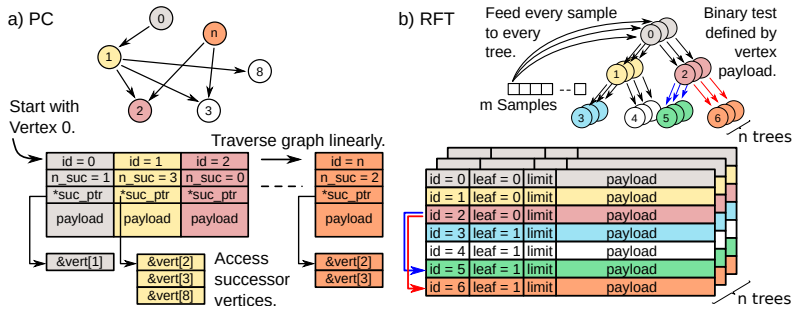

Fig. 8. The synthetic benchmark applications: a) Pointer Chasing (PC) and b) Random Forest Traversal (RFT).

Pointer Chasing (PC): This benchmark is a representative example for a wide variety of pointer-chasing applications from the graph processing domain [6, 11]. Prominent examples include PageRank (PR), breadth-first search, shortest path search, clustering, and nearest neighbor search. Due to the irregular and data-dependent access pattern to shared memory and low locality of reference, they represent worst-case scenarios for a virtual memory subsystem. The principle of the benchmark is visualized in Fig. 8 a). The host builds up the graph and stores the vertex information in a single array in main memory. Every array element stores the number of successors of the corresponding vertex, a pointer to an array of successor vertex pointers, and a configurable amount of payload data. For every vertex, the PMCA reads the number of successors and copies the payload data and successor pointers to the L1 SPM using DMA. Then, it performs a configurable number of

computation cycles on the payload data and writes the payload data to the successors in shared main memory.

Random Forest Traversal (RFT): This benchmark operates on regular, binary decisions trees of configurable size typically used for regression and classification [10]. The host generates the decision trees in virtual memory and passes virtual address pointers to both the root vertices and the samples to be classified to the accelerator. The trees themselves are stored in a large array as shown in Fig. 8 b). Every vertex or array element stores a configurable amount of payload data and a limit for the binary decision. When a sample arrives at a non-leaf vertex, the accelerator loads the corresponding payload data from shared memory using DMA, performs a configurable amount of computation cycles using the sample and the payload data, and selects the next vertex based on the outcome of the computation. Upon arriving at a leaf vertex, the index of the leaf vertex is stored.

Memory Copy (MC): This application features a highly regular access pattern to shared memory and is a representative example for streaming applications. It simply lets the host create a buffer of configurable size in virtual memory and then passes a pointer to the buffer to the PMCA. The PMCA uses DMA transfers to copy the buffer from shared memory into the L1 SPM at maximum bandwidth.

7.3 Miss Handling Cost

We used a synthetic benchmark application to profile the VMM library and compare it with host-based RAB management. The *host* allocates a large array and passes a pointer to this array to the PMCA. The PEs of the PMCA then issue read and write accesses to this array. To measure the execution time of the different sub-functions of the RAB miss handling routines, the application uses the internal performance counters of the PEs and the peripheral timers of the cluster.

On average, the handling of a RAB miss using the VMM library takes 450 PMCA clock cycles. Around 50% of this time are spent in memory accessing the PT. The reconfiguration of RAB slices (to access currently unmapped PT sections and to map the requested user-space page) accounts for 20% of the execution time. The remaining time is spent on various sub-tasks of the VMM library that can be executed internal to the cluster. These include computing addresses of the next PMD and PTE entries, checking access privileges, handling the RAB slice replacement strategy, and waking the sleeping PE up after handling the miss. Overall, handling a RAB miss directly on the PMCA using the proposed VMM library is 11 times faster than handling misses on the host.

7.4 Synthetic Benchmark Results

The three benchmarks were run using the parameters summarized in Table 1 to test the presented SVM subsystem under various workload conditions. The selected parameter sets represent a mix of real use cases [10, 11], problem sizes still bearable by embedded systems, and performance transition points of the evaluation platform[1].

Table 1. Benchmark parameters

Pointer Chasing	
#Vertices	10 K
Vertex Size [B]	44 - 2,060
#Cycles per Vertex	10 - 1,000

Random Forest Traversal	
#Tree Levels	5 - 16
Vertex Size [B]	28 - 1,036
#Cycles per Vertex	10 - 1,000

Memory Copy	
#Iterations	1 - 64
Data Size [KiB]	64 - 1,024

[1]For example, we found that graphs with larger vertex count do not notably change the results for PC as long as the total graph size is larger than the amount of memory that can be remapped with the available 32 TLB entries (128 KiB).

We selected a randomly distributed graph of type Erdős-Rényi for PC and configured RFT to sort random input numbers. As a consequence, access patterns to shared memory are highly irregular and randomized. We measured the execution time on the PMCA, both when using the PTW design presented in this work and when letting the host handle the RAB misses. We compare these two designs with an ideal SVM subsystem, i.e., a TLB with a single-cycle lookup latency that always contains the requested mapping and thus never misses. While ideal SVM would be desirable, even full-fledged hardware IOMMUs today are far from reaching this performance: a recent study has shown that their average performance on a mixed set of accelerator workloads is less than 13% of ideal SVM [8]. We modeled the ideal SVM subsystem by letting the host copy the shared data to a reserved, physically contiguous section and translate any virtual address pointers in the shared data. While this allows for zero overhead at PMCA run time, it leads to considerable design-time overheads and very high offload costs, which are to be avoided by using SVM.

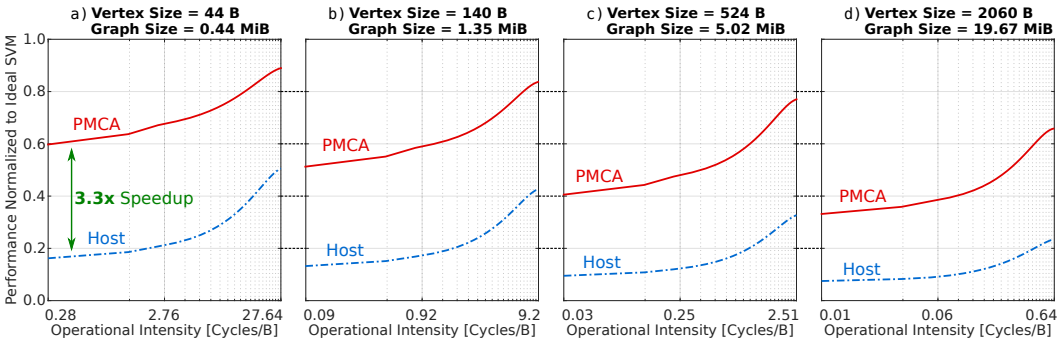

Fig. 9. PC results for different vertex/graph sizes and resulting operational intensities. The labels PMCA and Host denote the unit that manages the RAB in the corresponding curve.

Pointer Chasing (PC): Fig. 9 shows the performance normalized to ideal SVM for different numbers of computation cycles per vertex and increasing vertex sizes, i.e., the total amount of data in the linear array per vertex, in a) - d). Given a specific vertex size, varying the number of computation cycles per vertex changes the operational intensity defined as the number of operations performed per data transferred between shared main memory and the local SPM of the PMCA. The operational intensity is a characteristic property of a specific algorithm implementation. It is denoted on the x-axis of the plots. Since the graph cannot be remapped by the available TLB entries at once and since the access pattern is random, PC is dominated by RAB miss handling. Thus, letting the PMCA manage the RAB is always substantially better than involving the host. The proposed SVM system achieves between 60 and 88% of the performance of ideal SVM for operational intensities between 0.28 and 28 cycles per byte. For example, a single-precision floating-point implementation of PR performs 1 division, 1 multiplication per vertex and 1 addition per successor, and accesses 20 B of data for these operations. On the ARM Cortex-A9 host CPU, this leads to an intensity of 1.2 cycles/B [3].

Random Forest Traversal (RFT): The results for RFT are shown in Fig. 10. The x-axis of the plots denotes the number of tree levels. Every plot shows two pairs of curves corresponding to different operational intensities. It can be seen that changing the vertex size from 28 to 268 B (Fig. 10 a) to b)) mainly shifts the transition point beyond which letting the PMCA manage the RAB becomes increasingly beneficial. This transition point corresponds to where the total tree size becomes greater than the capacity of the TLB. For smaller tree sizes, the application causes only

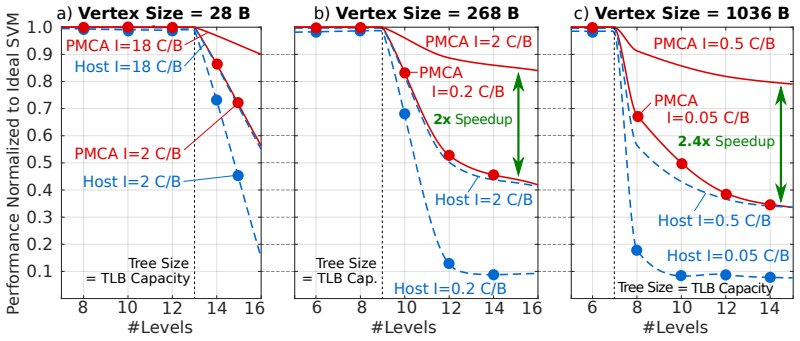

Fig. 10. RFT results for different vertex and tree sizes, and resulting operational intensities. The labels PMCA and Host denote the unit that manages the RAB in the corresponding curve.

compulsory TLB misses, i.e., TLB misses happening on the very first access to every memory page. Above the transition point, more and more TLB misses to previously mapped but at some point discarded pages (capacity misses) are generated until the performance saturates. In this regime, the proposed design is up to 2.4 times faster than when the host manages the RAB. Also, it can be seen that increasing the vertex size (Fig. 10 a) to c)) does not notably affect performance in the regime dominated by capacity misses. The distance between a vertex and its successors increases with the depth in the tree. As the size of all vertices in a tree level reaches the size of a memory page, accessing the successors of a vertex on the next level always generates a TLB miss. A larger vertex size means that more data on the remapped page is accessed and helps to amortize the cost of the miss despite a potentially very low operational intensity.

Memory Copy (MC): Fig. 11 shows the performance for MC normalized to the ideal SVM system for different input data sizes. In practice, the size of the L1 SPM might not suffice to hold the entire input data at once and multiple iterations over the same input data might be required to apply a specific application kernel. The number of iterations are denoted on the x-axis. For data sizes smaller than the TLB capacity, it can be seen that performing multiple iterations helps to amortize the compulsory TLB misses occurring during the first iteration. This effect is much more pronounced when the host is managing the TLB because misses are more expensive in this case.

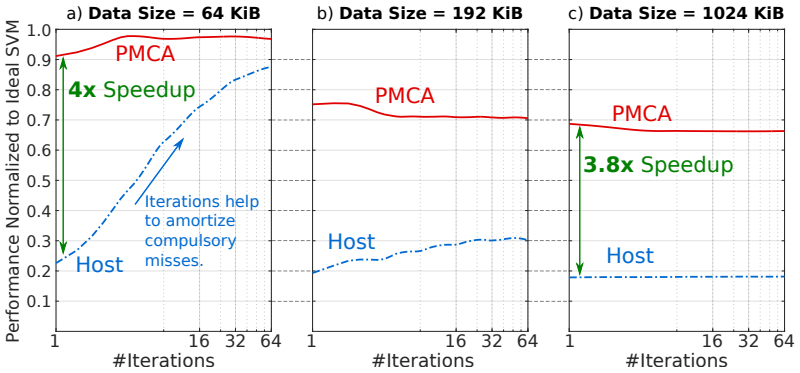

Fig. 11. MC results for different data sizes and variable number of iterations. The labels PMCA and Host denote the unit that manages the RAB in the corresponding curve.

As the data size becomes greater than the TLB capacity, this effect diminishes. It can also be seen that with increasing data size, the relative performance of the SVM schemes under comparison decreases. For larger data sizes, the number of DMA transfers increases while the size of the local SPM stays constant. Thus, also the total SVM overhead associated with the DMA transfers increases. Overall, letting the PMCA handle the TLB improves performance by factors of 3.8 to 4 compared with host-based management.

MC offers a highly regular access pattern to SVM that is known in advance. Our virtual memory management approach offers the flexibility to exploit this information on the host side at offload time for prefetching. Fig. 12 compares the achievable performance of such a scheme when copying a buffer with a size of 1 MiB. Clearly, this allows to improve performance beyond what is possible with purely reactive miss handling (MH). The achievable performance is very close to that of ideal SVM.

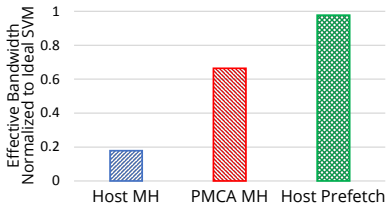

Fig. 12. MC effective bandwidth normalized to ideal SVM.

7.5 Real Application Benchmark Results

To evaluate the performance of the proposed SVM design for real-world applications operating on irregular, pointer-rich data structures, we have used three different applications that combine the access patterns of the synthetic benchmarks. PR [6] is a typical example for a PC application. Since the PMCA implementation of our evaluation platform does not feature floating-point units, we have used a custom fixed-point software library, which leads to an increased operational intensity. The vertices have a size of 20 B. The Random Hough Forest (RHF) application is part of the classification stage of an object detector [10]. Besides the actual classification stage exhibiting the access pattern of RHF on trees with 16 levels, also a set of filtering operations are executed on the PMCA. These filtering operations are well parallelizable, operate on the input images previously copied to the L1 SPM and are highly compute intensive. They can help to amortize potentially high overheads during phases operating on SVM. The transfer of the input image features an access pattern to shared memory very similar to MC. The third application performs face detection (FD) using the well-known Viola-Jones algorithm [30] operating on degenerate decision trees. The access pattern to shared memory is similar to RFT. The intensive computation of the integral and squared integral images fed to the detector is also performed on the PMCA using data in the L1 SPM. Similar to RHF, the transfer of the actual input image features a streaming-type access pattern like MC.

We have run the benchmarks for different data sets. Fig. 13 compares the measured performance normalized to ideal SVM when using the host or the PMCA to manage the RAB. For PR with 4,000 vertices as well as for FD, the pointer-rich data structure can be remapped completely with the available RAB slices. Only few compulsory misses must be handled, which can be well amortized by the computation, independent of whether the PMCA or the host manages the RAB. As the number of vertices for PR increases to 10,000, the overall execution time starts to get dominated by capacity misses. Letting the PMCA manage the RAB improves the performance by a factor of

roughly 1.5. For RHFs, the highly compute-intensive filter operations allow to partially amortize the many capacity misses during the classification phase (1.75 MiB of data per tree). As the number of trees and thus the number of capacity misses increases, more and more time is spent on handling these capacity misses while the amount of filter operations remains constant. Letting the PMCA manage the RAB using the VMM library presented in this work brings the performance close to the performance of ideal SVM.

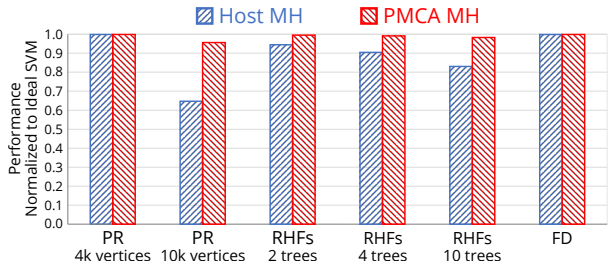

Fig. 13. Results for real-world applications normalized to ideal SVM.

8 CONCLUSION

In this work, we have presented a novel hardware-software solution for accelerator-managed shared virtual memory in heterogeneous embedded systems on chip. Our solution is based on an accelerator-side helper thread using a virtual memory management software library that enables the accelerator to autonomously perform the correctness- and performance-critical task of operating its virtual memory hardware, including page table walking and IOTLB management. The library provides a simple API and is compatible with any host CPU architecture supported by the Linux kernel. In addition, the design retains the flexibility of a software-managed solution, which allows, e.g., the exploration of novel ideas such as managing the IOTLB in a collaboration among host and accelerator. We have validated the design on our evaluation platform using both parametrizable, synthetic benchmarks and real-world benchmarks covering various application domains including linked data structures. Letting the accelerator manage its VM hardware autonomously eliminates frequent interventions of the host CPU and brings substantial performance improvements by factors of up to 3.8 compared with state-of-the-art host-based IOTLB management. Compared with an ideal system for SVM, the performance lies within 50% for purely memory-bound applications and within 5% for real applications.

We are extending our studies in multiple directions, including the dynamic parallelization of the VMM library to reduce the miss latency when many misses occur simultaneously. Also, we are investigating alternative and larger RAB designs and smart prefetching techniques to reduce the RAB miss rate. Finally, we are extending the entire SVM framework to support the operation of multiple many-core clusters combined with a state-of-the-art ARMv8 host processor.

REFERENCES

[1] Adapteva Inc. Parallela Reference Manual. Technical reference manual. (2014).
[2] AMD Inc. AMD Compute Cores. White Paper. (2014). www.amd.com/documents/compute_cores_whitepaper.pdf.
[3] ARM Ltd. Cortex-A9 Floating-Point Unit. Technical reference manual. (2012).
[4] ARM Ltd. AMBA AXI and ACE Protocol Specification. Protocol specification. (2013).
[5] ARM Ltd. ARM CoreLink MMU-500 System Memory Management Unit. Technical reference manual. (2016).
[6] S. Brin and L. Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *IW3C-7*. 107–117.
[7] Y. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei. 2016. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. In *DAC-53*. 109:1–109:6.
[8] J. Cong, Z. Fang, Y. Hao, and G. Reinman. 2017. Supporting Address Translation for Accelerator-Centric Architectures. In *HPCA-23*. 37–48.
[9] J. Corbet. Fixing the contiguous memory allocator. LWN article. (2015). http://lwn.net/Articles/636234/.
[10] J. Gall and V. Lempitsky. 2009. Class-Specific Hough Forests for Object Detection. In *CVPR-27*. 1022–1029.
[11] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. 2014. How Well do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis. In *IPDPS-28*. 395–404.
[12] HSA Foundation. HSA Foundation. (2012). www.hsafoundation.com.
[13] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *ICCD-34*. 25–32.
[14] Intel Corp. The compute architecture of Intel Processor Graphics Gen9. White Paper. (2015). https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf.
[15] Intel Corp. Arria 10 Device Overview. Product Specification. (2016).
[16] Kalray S.A. MPPA MANYCORE. (2014).
[17] G. Kornaros, K. Harteros, I. Christoforakis, and M. Astrinaki. 2014. I/O Virtualization Utilizing an Efficient Hardware System-Level Memory Management Unit. In *ISSoC '14*. 1–4.
[18] A. Kurth, A. Tretter, P. A. Hager, S. Sanabria, O. Göksel, L. Thiele, and L. Benini. 2016. Mobile Ultrasound Imaging on Heterogeneous Multi-Core Platforms. In *ESTIMedia-14*. 9–18.
[19] M. Lavasani, H. Angepat, and D. Chiou. 2014. An FPGA-based In-Line Accelerator for Memcached. *IEEE CAL* 13, 2 (2014), 57–60.
[20] Y. Li, R. Melhem, and A. K. Jones. 2013. PS-TLB: Leveraging Page Classification Information for Fast, Scalable and Efficient Translation for Future CMPs. *ACM TACO* 9, 4 (2013), 28:1–28:21.
[21] P. Mantovani, E. G. Cota, C. Pilato, G. Di Guglielmo, and L. P. Carloni. 2016. Handling Large Data Sets for High-performance Embedded Applications in Heterogeneous Systems-on-chip. In *CASES '16*. 3:1–3:10.
[22] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit. 2012. Platform 2012, a Many-core Computing Accelerator for Embedded SoCs. In *DAC-49*. 1137–1142.
[23] M. Nazarewicz. A deep dive into CMA. LWN article. (2012). http://lwn.net/Articles/486301/.
[24] S. Park, M. Kim, and H. Y. Yeom. 2016. GCMA: Guaranteed Contiguous Memory Allocator. *SIGBED Rev.* 13, 1 (2016), 29–34.
[25] O. Peleg, A. Morrison, B. Serebrin, and D. Tsafrir. 2015. Utilizing the IOMMU Scalably. In *USENIX ATC '15*. 549–562.
[26] B. Pichai, L. Hsu, and A. Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs. In *ASPLOS-19*. 743–758.
[27] J. Power, M. D. Hill, and D. A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *HPCA-20*. 568–578.
[28] D. Rossi, I. Loi, F. Conti, G. Tagliavini, A. Pullini, and A. Marongiu. 2014. Energy efficient parallel computing on the PULP platform with support for OpenMP. In *ICEEEI-28*. 1–5.
[29] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. 2015. CAPI: A Coherent Accelerator Processor Interface. *IBM J. Res. Dev.* 59, 1 (2015), 7:1–7:7.
[30] P. Viola and M. Jones. 2004. Robust Real-Time Face Detection. *IJCV* 57, 2 (2004), 137–154.
[31] P. Vogel, A. Marongiu, and L. Benini. 2017. Lightweight Virtual Memory Support for Zero-Copy Sharing of Pointer-Rich Data Structures in Heterogeneous Embedded SoCs. *IEEE TPDS* 28, 7 (2017), 1947–1959.
[32] Xilinx Inc. Zynq-7000 All Programmable SoC Overview. Product Specification. (2016).
[33] Xilinx Inc. SDSoC Environment User Guide. User Guide. (2017). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1027-sdsoc-user-guide.pdf.
[34] Xilinx Inc. Zynq UltraScale+ MPSoC Data Sheet: Overview. Advance Product Specification. (2017).

Received April 2017; revised June 2017; accepted June 2017