Scalable and Efficient Virtual Memory Sharing in Heterogeneous SoCs with TLB Prefetching and MMU-Aware DMA Engine

This is the post peer-review accepted manuscript of:

A. Kurth, P. Vogel, A. Marongiu, and L. Benini, "Scalable and Efficient Virtual Memory Sharing in Heterogeneous SoCs with TLB Prefetching and MMU-Aware DMA Engine", 36th IEEE International Conference on Computer Design (ICCD 2018), Orlando, FL, USA, October 7-10 2018.

The published version is available online at: https://doi.org/10.1109/ICCD.2018.00052

Scalable and Efficient Virtual Memory Sharing in Heterogeneous SoCs with TLB Prefetching and MMU-Aware DMA Engine

Andreas Kurth*, Pirmin Vogel*, Andrea Marongiu†*, and Luca Benini*†
*Integrated Systems Laboratory, ETH Zurich, Switzerland
lastname@iis.ee.ethz.ch
†University of Bologna, Italy

***Abstract*—Shared virtual memory (SVM) is key in heterogeneous systems on chip (SoCs), which combine a general-purpose host processor with a many-core accelerator, both for programmability and to avoid data duplication. However, SVM can bring a significant run time overhead when translation lookaside buffer (TLB) entries are missing. Moreover, allowing DMA burst transfers to write SVM traditionally requires buffers to absorb transfers that miss in the TLB. These buffers have to be overprovisioned for the maximum burst size, wasting precious on-chip memory, and stall all SVM accesses once they are full, hampering the scalability of parallel accelerators.**

In this work, we present our SVM solution that avoids the majority of TLB misses with prefetching, supports parallel burst DMA transfers without additional buffers, and can be scaled with the workload and number of parallel processors. Our solution is based on three novel concepts: To minimize the rate of TLB misses, the TLB is proactively filled by compiler-generated Prefetching Helper Threads, which use run-time information to issue timely prefetches. To reduce the latency of TLB misses, misses are handled by a variable number of parallel Miss Handling Helper Threads. To support parallel burst DMA transfers to SVM without additional buffers, we add lightweight hardware to a standard DMA engine to detect and react to TLB misses. Compared to the state of the art, our work improves accelerator performance for memory-intensive kernels by up to 4× and by up to 60 % for irregular and regular memory access patterns, respectively.

I. Introduction

In heterogeneous systems on chip (HeSoCs), a general-purpose multicore CPU, the *host*, is co-integrated with programmable many-core accelerators (PMCAs) on a single die. This design holds the promise of combining the versatility of the host CPU with the energy efficiency and computing performance of the highly parallel accelerators.

One of the major difficulties in programming HeSoCs is having to explicitly manage the multi-level, non-uniform memory system. On the host, coherent caches and virtual addresses make the memory hierarchy completely transparent to the application programmer. On the PMCA, however, scratchpad memories (SPMs) are often preferred to hardware-managed caches for the implementation of on-chip memory hierarchies. SPMs are physically addressed and data transfers to and from them are controlled by software, preferably using direct memory access (DMA) transfers.

This work was partially funded by the EU's H2020 projects HERCULES (No. 688860) and OPRECOMP (No. 732631).

To alleviate this difficulty and to enable sharing of linked data structures, the Heterogeneous System Architecture Foundation [1] pushed an architectural model where host and PMCAs communicate via coherent shared virtual memory (SVM). For coherency with the host data caches, most SoCs today offer accelerators access to coherent interconnects [2], [3]. For the translation of virtual addresses, there are two main approaches: In the all-hardware approach followed by many embedded SoC vendors, a full-fledged input/output memory management unit (IOMMU) translates addresses autonomously [4], [5]. It is comprised of a translation lookaside buffer (TLB), parallel hardware page table walkers (PTWs), transaction and data buffers, and coherent page table caches. The alternative is a hybrid hardware-software design, which consists of a TLB controlled by software (e.g., by a kernel driver on the host [6], [7] or directly by the accelerator [8]). We subsequently refer to the former class of IOMMUs as *conventional* and to the second class as *hybrid*. While conventional IOMMUs have the advantage of being transparent to the PMCAs and of offering the minimal latency for handling an isolated TLB miss, they have three significant drawbacks:

First, parallel, interleaved accesses by PMCAs to independent virtual addresses require parallel PTWs. While the number of parallel accesses is a time-variant run-time property of programs executed by the PMCAs, the number of PTWs is a fixed design parameter in conventional IOMMUs. To accommodate a wide range of parallel workloads, the number of parallel hardware PTWs must be overprovisioned, wasting hardware resources in most use cases.

Second, enabling DMA engines to access SVM in conventional IOMMUs requires a data buffer in the IOMMU to absorb write bursts to addresses that miss in the TLB. This buffer requires a significant amount of memory: it must have at least the size of the largest DMA burst, and is usually even larger because no more SVM accesses (by *any* master, not just the missing DMA engine) can be processed once (and as long as) that buffer is full.

Third, a conventional IOMMU manages its TLB purely reactively: new entries are set up only after a TLB miss. While some IOMMUs [9], [5] can speculate on future memory accesses based on past access patterns, doing so is very inaccurate for nonlinear, interleaved access patterns and negatively

Accepted for publication in *Proceedings of the 36th IEEE International Conference on Computer Design (ICCD)*, October 7-10, 2018, Orlando, FL, USA.

affects performance [9]. Misses can thus occur frequently, and high-performance conventional IOMMUs include coherent data caches for page table entries [5] to reduce the latency of handling a TLB miss. If the TLB was managed by software threads in the PMCA instead (as in hybrid IOMMUs), it would be possible for those threads to set up TLB entries ahead of time based on run-time information inside the PMCA. Research on data caches [10], [11] has long shown that prefetchers that know the running program and its status can be far more accurate than those that can only see a stream of memory addresses.

The hybrid design, on the other hand, theoretically does not have these drawbacks. However, the state-of-the-art implementation [8] features only a single PTW thread, only supports DMA bursts that are guaranteed not to miss (e.g., by locking the corresponding TLB entries), and does not perform any prefetching.

In this work, we resolve these limitations. To the best of our knowledge, this work is the first to:

1) implement accurate, compiler-generated prefetching for a shared TLB (§ IV-A), which significantly reduces the rate of TLB misses,
2) offer a flexible number of parallel TLB miss handlers (§ IV-B), which keeps the miss handling latency constant for scalable parallel workloads, and
3) offer shared virtual memory accessible by DMA transfers without additional buffers (§ IV-C).

Compared to the state-of-the-art hybrid IOMMU [8], our contributions improve the PMCA performance for memory-intensive kernels by up to 4× and by up to 60 % for irregular and regular memory access patterns, respectively (§ V-C). Compared to using data buffers to absorb bursts from DMA engines, our solution requires two orders of magnitude less memory (§ V-D) and scales better, as it only stalls the missing DMA engine.

II. Related Work

The vast majority of commercial systems today features conventional IOMMUs [5], [12], [13], [14] to completely abstract the SVM implementation from the PMCAs. While simple to use, that approach is limited in scalability (handling parallel misses and absorbing burst transfers) and efficiency (reactive TLB management).

a) Parallel TLB miss handling and page table walking: The fixed number of shared hardware PTWs puts an upper bound on the scalability of conventional IOMMUs to parallel accelerators. A recent study [9] has shown that an integrated GPU with 8 parallel compute units (CUs) quickly saturates the miss handling capabilities of an IOMMU with 16 hardware PTWs, after which the GPU's performance becomes bounded by TLB miss handling latency. To avoid this, the current proposal for address translation on GPUs [15], [4] is to add one MMU before the L1 cache in every CU. Each such CU MMU has its private TLB and either has its own PTW [4] or shares a highly-threaded PTW [15]. As this approach adds a significant amount of hardware, its parameters (e.g., TLB size, number of PTWs) must be carefully balanced *at hardware design time* to neither present a bottleneck for SVM-heavy applications nor reduce the compute-per-area ratio for applications that use SVM in a lighter way. The miss handling throughput of hybrid IOMMUs, where page table walks are performed by software threads, on the other hand, can be scaled *at run time* by scheduling PTWs when required. However, efficiently managing a TLB shared by many parallel processing elements (PEs) and notifying individual PE with low latency about handled misses is not trivial. For this reason, current hybrid SVM solutions [8], [6] only feature a single PTW thread. In this work, we show how to efficiently manage a shared TLB with multiple software PTW threads.

b) Handling bursts missing in the TLB: The buffers in conventional IOMMUs that absorb write bursts missing in the TLB [5] are another limiting factor for accelerators based on DMA transfers: When (and as long as) the limit on outstanding misses is reached, the IOMMU cannot translate any further transactions, even if they would hit. This creates backpressure from the IOMMU slave port to the connected master ports, stalling each master port on its next SVM access. Hybrid IOMMUs, on the other hand, signal the TLB miss back to the master and drop the transaction [7]. This allows to handle misses on a shared TLB in a much more scalable way: instead of creating congestion on shared resources (e.g., buffers in the IOMMU, interconnect), the transaction that missed stays in the source memory, keeping shared resources clear for other accesses. To support this, the DMA engine must be able to keep track of bursts that missed and reissue them when the miss has been handled. In this work, we introduce a lightweight hardware extension that adds this feature for a standard DMA engine.

c) Reducing TLB misses through prefetching: Reducing the number of TLB misses is another effective way to reduce the run time overhead of SVM, orthogonal to reducing the miss handling latency. There are two independent strategies to achieve this: The first is to increase the capacity of the TLB, for which both conventional and hybrid IOMMUs feature multi-level TLBs [5], [7]. The second is to ensure the timelineness of TLB entries, e.g. through prefetching. Prefetching for shared TLBs is not yet well-understood: Some conventional IOMMUs feature a very simple prefetcher, which adds two subsequent pages to the TLB in case of a miss to the first [5]. However, this prefetcher is deactivated by default because it harms performance in most cases [5]. Prefetching is also supported by the PCIe Address Translation Services [16], but a recent study [9] examined a benchmark with low locality, found that having a GPU prefetch eight contiguous pages degrades performance by up to 3×, and concluded that research on application-aware prefetching is required. In this work, we design and implement accurate prefetching for a shared TLB, which significantly reduces the rate of TLB misses. We focus on linked data structures (LDSes), which are the predominant source of scattered memory accesses in many programs. Our design is inspired by the following prefetchers for data caches.

Prefetchers that get information about the running program from software [10], [17], [18], [11], [19], [20], [21], [22] are far more effective than heuristic hardware units [23], [24]

for LDSes: Heuristic hardware prefetchers for LDSes identify pointers as they are loaded from memory, prefetch their content before they are dereferenced, and store it in a separate pointer cache [24] or in the data cache of the processor [23]. All pointers identified by the heuristic hardware are prefetched recursively, which leads to a low prefetch accuracy, consequently polluting the cache and decreasing performance in many cases. To improve prefetch accuracy, the hardware prefetcher in hybrid hardware/software prefetchers [20], [11] is controlled by software, e.g., from the main processor through special instructions to identify useful prefetches [20] or by running a separate prefetching program [11]. Prefetch code can be written manually by a developer [10], [19] or generated automatically by a compiler [18], [17], [21], [22], through static or dynamic profiling or both. Compilers can accurately identify pointers and prioritize their prefetching according to dependencies. Once prefetchers for LDSes are accurate, their effectiveness is limited by memory latency, as prefetch targets in LDSes depend on earlier pointer dereferences. As a dedicated hardware prefetcher co-located with the processor has the same memory latency as the processor itself, pure software prefetchers have been explored instead [19], [17], [21], [22]. Prefetches inserted inline with the actual program code [17], however, are limited to targets that are known when that line of code is executed. Otherwise, the actual program has to be stalled while the pointers leading to the prefetch target are followed. A promising alternative is to run an additional prefetching thread on the same multithreaded processor core [25], [19] or on another core in the same processor [21], [22]. Another important advantage of these separate prefetcher threads is that their throughput can be scaled to the demands of the application at compile-time or at run-time or both, especially for the high degree of parallelism offered by PMCAs. A key difficulty of software prefetches executed by another thread is the timeliness of the prefetches, which is why prefetching threads have primarily been explored for coarse-grained prefetching [22], [21].

While there are a number of works using heuristic hardware units for TLB prefetching [26], [27], [28], this work (to the best of our knowledge) is the first to use compiler-generated software threads for TLB prefetching. For the first time, this allows to accurately prefetch TLB entries for LDSes. Compared to the related compiler-generated software prefetchers, our solution is novel in how it issues fine-grained, timely prefetches into a shared resource (the TLB) in a scalable way without causing negative interference.

III. Target Architecture Template

The heterogeneous system targeted in this work combines two architecturally different processors in a single chip. As shown in Fig. 1, the HeSoC is composed of a general-purpose multi-core CPU (the host) and a domain-specific PMCA. The host CPU features a cache-coherent memory hierarchy, runs a full-fledged operating system, and manages inputs and outputs of the HeSoC. The PMCA complements the host by offering high computational performance and efficiency for specific application domains.

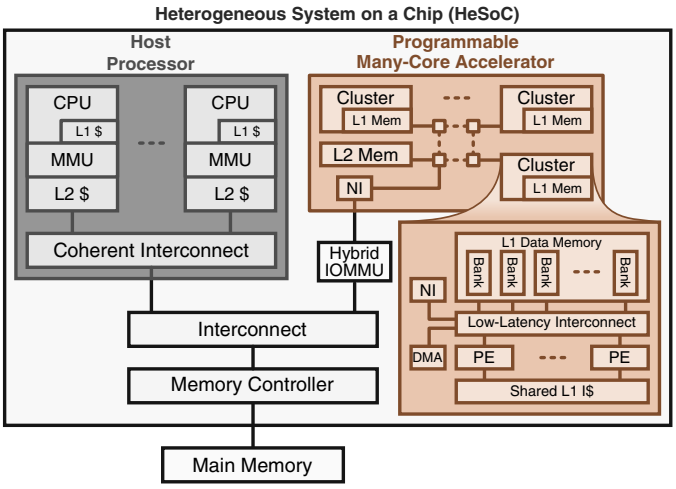

Fig. 1. Template of the target architecture.

The PMCA we consider uses a multi-cluster design [29], [30] to enable architectural scaling. In each cluster [31], multiple PEs share an L1 data SPM and an L1 instruction cache [32], both multi-banked (twice as many banks as PEs), through a low-latency logarithmic interconnect [33]. Multiple clusters are attached to the main network of the PMCA, through which they share an L2 SPM.

The off-chip main dynamic random access memory (DRAM) is physically shared by the host and the PMCA. To exploit data locality, both host and PMCA keep the most frequently accessed data in fast, local storage of their internal memory hierarchy. While the host relies on hardware-managed caches, the PMCA uses multi-channel, high-bandwidth DMA engines and double-buffering schemes to overlap data movement with computation on data in the L1 SPMs. The widely-adopted AXI4 protocol is used between host and PMCA and on the network in the PMCA.

The IOMMU allows the PMCA to share the virtual memory space of an application running on the host. It is a hybrid design [7], consisting of a TLB of configurable size completely managed by software running on the PMCA. The TLB is fully associative and processes look-ups within a single clock cycle. In case of a TLB miss, the IOMMU stores the metadata in a hardware queue, responds with an error, drops the transaction, and processes the next one. In case of a TLB hit, the IOMMU translates the virtual address to a physical one, forwards the transaction through the master port, and processes the next transaction.

One PE of the PMCA manages the TLB in the IOMMU. Upon a TLB miss, it reads the metadata of the missing transaction from the hardware queue in the IOMMU, walks the page table of the offloaded process, replaces an older TLB entry with the new translation, and notifies the PE that encountered the miss, which then retries the memory access. As the memory access latency in page table walks dominates the miss handling latency, this software PTW has about the same latency as a dedicated hardware PTW [8].

PEs within a cluster execute in a single program multiple data (SPMD) fashion and share a multi-ported, multi-banked

instruction cache [33]. They can exchange data with low latency and low congestion through the L1 data memory, which also offers an atomic *test-and-set* read-modify-write operation. A dedicated event unit within the cluster supports interrupts, barriers, and software-triggered events for low-overhead synchronization.

Each cluster includes a DMA engine optimized both in throughput and area for transfers from or to the cluster's tightly-coupled SPMs [34]. It supports up to 16 outstanding AXI4 bursts with only minimal internal buffers thanks to the low-latency connection to the SPMs. Each PE has a private command interface on the DMA engine, which allows multiple PEs to simultaneously enqueue DMA transfers without the need for synchronization. The control unit of the DMA engine internally arbitrates between the per-PE command interfaces. PEs can enqueue coarse-grained transfer commands (up to 64 KiB), which are split up by the control unit into fine-grained bursts (up to 2 KiB) to meet alignment requirements and to facilitate time-multiplexing of downstream AXI resources. As soon as a coarse-grained transfer is complete, the DMA engine notifies the PE via the event unit.

IV. Implementation

In this section, we detail our compiler-generated TLB prefetchers, which significantly reduce the rate of TLB misses (§ IV-A), our scalable multi-threaded TLB miss handlers, which keep the TLB miss handling latency constant for scalable parallel workloads (§ IV-B), and our hybrid-IOMMU-capable DMA engine, which can handle TLB misses without additional data buffers (§ IV-C).

A. Helper Thread Prefetching

As TLB miss handlers are dominated by memory latency, frequent TLB misses inevitably entail a large run time overhead. As a consequence, managing the TLB solely by reacting on misses is not sufficient. Instead, TLB entries could be set up ahead of the instant they are used in a prefetching manner.

The following observations motivate our prefetcher design:

- It shall not rely solely on run-time information (e.g., memory content, memory access patterns). This is the black-box approach taken by hardware prefetchers, which is not accurate for LDSes.
- It shall not rely solely on compile-time information (e.g., algorithms, data structures) because this neglects all dynamic information (e.g., data-dependent memory accesses, delays due to interference) required for timely prefetches.
- It shall be portable across applications. While software prefetches can be inserted manually, doing so effectively requires in-depth knowledge of the target platform and laborious analysis of the application. Prefetch insertion shall be fully automatic, not burdening developers.
- It shall exploit the cluster architecture of the PMCA, where tightly-coupled L1 SPM allows to share the state of PEs with low latency and little interference.
- Its prefetching throughput shall be scalable at compile-time (because different programs have different memory access patterns and intensities) and at run-time (due to phase-based program behavior [35]).

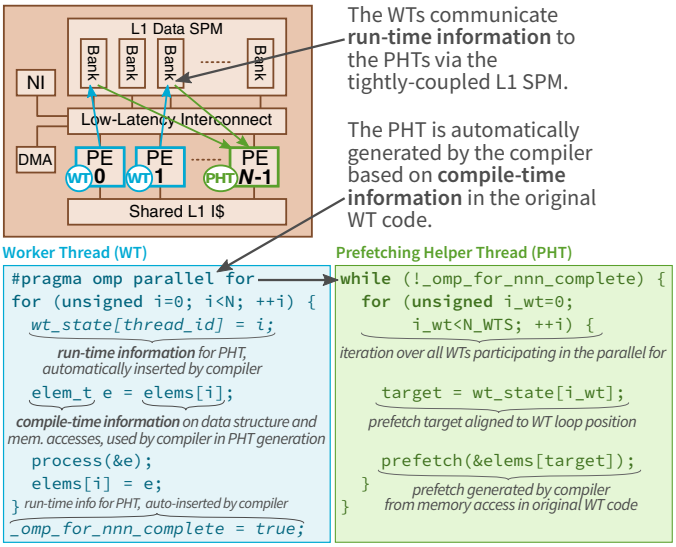

Fig. 2. The concept of prefetching with closely-coupled helper threads using the example of a very simple parallel `for` loop.

Combined, these considerations led us to the concept of prefetching with closely-coupled helper threads shown in Fig. 2. Our execution model assumes that part of the PEs in a PMCA cluster are statically allocated to executing the original application workload. The workload is distributed among as many *Worker Threads (WTs)* according to the semantics of parallel programming models such as OpenMP [36]. The remaining PEs in the cluster are statically allocated to execute *Prefetching Helper Threads (PHTs)*, which our compiler automatically generates by stripping down the code of the WTs: The idea is to remove all statements that do not access SVM and do not determine the address or the occurrence of an SVM access, and to replace SVM accesses in the remaining code with a call to a prefetch method. Additionally, the compiler inserts store instructions to the L1 SPM into the WT to share its state of execution and load instructions into the PHT to read the execution state. The prefetch method does not modify the TLB itself; rather, it checks if a page is currently in the TLB and, if it is not, informs the standard TLB miss handlers (through the queue of TLB misses) that a TLB entry must be set up ahead of the moment when a worker thread actually requires the data on a page. Prefetching TLB accesses are described in more detail in § IV-A2.

Prefetches are issued conditionally on the current position of the WT relative to the current position of the PHT. For example, a fixed window wherein prefetches are issued can be defined: Assume w_k is the position of WT k in a parallel loop and d and D are the minimum and maximum prefetching distances, respectively. Then the PHT has to make sure that the position of its next prefetch for WT k, p_k, fulfills $w_k + d \leq p_k \leq w_k + D$. If $p_k > w_k + D$, then the PHT is ahead of WT k by more than the maximum prefetching distance and the PHT will not issue a prefetch. If $p_k < w_k + d$, then the PHT is behind the minimum prefetching distance and the PHT will set p_k to a

position inside the window. When p_k is inside the window, the PHT will prefetch at p_k and then increment p_k.

1) Compiler Algorithm to Generate PHTs: The compiler algorithm to generate a PHT from the code for a WT comprises two stages: The first stage recursively traverses the abstract syntax tree (AST) of the body compound of the WT. In a forward pass, a data dependency graph (DDG) for each SVM variable (i.e., a variable that dereferences a pointer to SVM, possibly through other variables and pointers) is constructed. In a backward pass, memory accesses to DDG leaf nodes are rewritten as prefetches and all statements that are not in the DDG of an SVM variable are removed. The second stage recursively traverses the modified AST and prunes redundant prefetches.

The forward pass is recursively invoked on an AST node and scope tuple, where the scope is a list of variables together with their DDG. It creates the PHT for the given AST node by constructing a DDG for each SVM variable and invoking the backward pass afterwards. The forward pass essentially distinguishes two classes of AST nodes: Declarations extend the scope of the subsequent nodes, and assignments modify the DDG of their left-hand side variable. Compounds establish a local scope, within which the children of the compound are first modified in forward order by the forward pass, then in backward order by the backward pass.

The backward pass is also invoked on an AST node and scope tuple. Based on the DDG of each variable, it rewrites dereferences of SVM pointers that are leaf nodes into prefetches and removes all statements that are not in the DDG of an SVM variable. It distinguishes three classes of AST nodes: Conditionals and loops add a control flow dependency to variables they reference. Declarations remove the declared DDG node from the scope (since the variable is undeclared before the declaration). Finally, assignments are either replaced by prefetches, left intact, or dropped completely, based on whether their left- and right-hand sides contain SVM variables.

2) Hardware Requirements: A prefetch load or store is slightly different from a regular memory access: upon a hit in the TLB, a prefetch transaction must not be forwarded downstream the memory hierarchy but instead be directly replied by the IOMMU as hit (with don't-care data in case of a load, since data returned by prefetch loads is ignored). A prefetch that misses in the TLB is not different from a regular miss to the hybrid IOMMU: it responds with a miss and drops the transaction. Conventional IOMMUs cannot support prefetches, since they lack the possibility to drop transactions and the masters using them lack the support for reacting to miss responses. Whether a load or store is a prefetch can be determined by a single bit sent with the request. Our implementation uses one bit in the AXI4 *user* field.

B. Multi-Threaded TLB Miss Handling

In the original implementation of the hybrid IOMMU we are using, TLB misses (only metadata) were enqueued by the IOMMU in a hardware queue [8]. This leftover from conventional IOMMUs presented a centralized bottleneck not required by the design, so we removed it and instead let PEs add an entry to a software queue located in the L1 data memory of their cluster upon a TLB miss. This atomic queue supports multiple parallel consumers and producers, and we implemented the atomicity with one enqueue mutex and one dequeue mutex based on the test-and-set functionality of the L1 memory.

For algorithms that make heavy use of SVM (especially those processing LDSes), a single miss handling thread (MHT) cannot cope with the rate at which WTs enqueue misses. In this case, the rate at which TLB misses are handled becomes the bottleneck. As an MHT is dominated by memory latency of the page table walking steps, the way to increase the miss handling rate is to let multiple MHTs work on different pages in parallel.

Two aspects are central for the design of the parallel MHTs: (1) given the sequence of misses in the queue, which MHT handles which miss, and (2) which MHT modifies which TLB entry.

For distributing misses among the MHTs, the simplest approach would be to let each MHT dequeue a miss, walk the page table, reconfigure a TLB entry, and wake up the PE that enqueued the miss. However, as two subsequent misses frequently go to the same page due to data locality (for an individual PE, e.g., with DMA bursts, but also for multiple PEs with shared data), this approach is not effective: Whenever a MHT dequeues a miss to a page that another MHT is already working on, it wastes run time and memory bandwidth on a redundant page table walk, and it wastes TLB capacity by setting up a redundant entry. Ideally, each MHT would dequeue all misses on the same page, walk the page table, and then wake up all PEs waiting for that page. However, this would require each MHT to traverse the entire miss queue (which can contain dozens of entries), locking both mutexes while it rearranges the queue without the misses to that page. This can take hundreds of clock cycles, during which no other PE can enqueue or dequeue misses.

Neither wasteful redundant miss handling nor making the miss queue a sequential bottleneck are acceptable, and our design avoids both: The MHTs share their state, i.e., which page each MHT is currently working on and which PEs it is going to wake up, through one word per MHT in the L1 data memory. When MHT A dequeues a miss, it first checks if another MHT is already working on the same page. If so, A tells the other MHT to also wake up the PE that caused the miss A just dequeued and dequeues another miss. If no other MHT works on the same page, A performs a prefetch memory access to the page to check whether the page has not been mapped since the miss. If the prefetch misses, A sets its state to that page and walks the page table. When A is done, it reads its state (which may have been updated in the meantime by other MHTs) and wakes up all assigned PEs.

Modifying a TLB entry takes two writes because virtual and physical page frame number together are longer than one data word. To avoid inconsistencies, the MHTs must thus ensure mutual exclusion when modifying a TLB entry. Any two different TLB slots are indepedent, though, so an MHT

should not preclude another from simultaneously modifying a different slot. As both TLB levels are highly associative, MHTs have multiple options for the placement of each TLB entry. To make effective use of associativity, the MHTs should agree upon one replacement order per set. These three requirements can be met by using one atomic counter per TLB set, located in memory shared by all MHTs, which determines the index of the entry to be replaced next in a set. An MHT determines the set number from the virtual page frame number, increments the atomic counter of that set, and modifies the entry at the index returned by the counter. If the number of MHTs is comparable to the nmuber of entries per set, the MHT must additionally lock the entry it modifies.

C. Hybrid-IOMMU-Capable DMA Engine

A hybrid IOMMU requires all masters that use it to be capable of tolerating TLB misses and keep track of which transactions missed. The DMA engine, however, was originally not designed to deal with error responses in a recoverable way and reported a transfer as complete as soon as it had received the final read or write response of the last burst, regardless of whether all bursts were successful or not. Thus, when a PE saw the completion of a transfer it started, it had no way to tell whether all data read or written were valid at the destination. To guarantee data integrity, all TLB entries required for the completion of a transfer (which can touch up to 17 4 KiB pages) had to be locked before the transfer could be programmed to the DMA engine and unlocked after it had completed. As the TLB is shared by multiple clusters, this limited the number of DMA transfers that could be enqueued at a given time and substantially reduced the effective data transfer bandwidth.

If the DMA engine can keep track of bursts that missed in the TLB and restart them after the miss has been handled, DMA transfers through the hybrid IOMMU can be much more efficient and scalable: An AXI burst may not cross a page, so each burst requires exactly one TLB entry at the instant its request arrives at the IOMMU. Requests of consecutive bursts can arrive back-to-back at the IOMMU, so multiple TLB entries need to be present only for a short time interval for an entire transfer spanning multiple pages to succeed.

To make the DMA engine compatible with the hybrid IOMMU, we designed and implemented a *retirement buffer* that keeps track of in-flight bursts. An entry in the buffer contains all metadata required to uniquely identify and reissue a burst: cluster-external and -internal address, length, AXI ID, DMA transfer ID, and whether it was a read or a write. When the AXI interface of the DMA engine sends a request, it adds a new entry to the retirement buffer, and when it receives the final response of a burst, it reports the success or failure of the burst with the responded AXI ID to the retirement buffer.

The retirement buffer must keep the order in which bursts were issued (because AXI bursts with the same ID are ordered) and must be able to complete bursts with different AXI IDs in any order. For these reasons, the retirement buffer can not be a simple FIFO queue. An alternative would be to have one FIFO queue per AXI ID, but this would waste hardware since every queue would need to have the capacity to store the maximum number of in-flight transfers.

addresses		length	IDs		R/W	state	next
external	internal	[Byte]	AXI	DMA			
						Free	
0x2462 3...	(...)	(...)	5	2	0	Reissuable	
0x2468 0ff0	0x3 55c0	16	7	1	0	Failed	
0x2462 3...	(...)	(...)	6	4	1	Reissuable	
0x2497 1...	(...)	(...)	1	3	1	In Flight	←tail
0x2468 1000	0x3 55d0	240	0	1	0	Peeked	
0x2468 0ef0	0x3 54c0	256	4	1	0	Peeked	←head
						Free	

Fig. 3. Organization of the burst retirement buffer.

Instead, our retirement buffer is a linked list implemented in hardware as shown in Fig. 3: The list entries are stored in a small register file that has as many words as the maximum number of in-flight transfers. Every word is wide enough to store the burst metadata mentioned above, the state of the entry (free, in-flight, failed, peeked, reissuable), and the index of the next burst entry. Additionally, the retirement buffer stores the index of the head, where order-preserving peek and pop operations start, and of the tail, where a new in-flight transfer gets enqueued.

The retirement buffer has three main interfaces: one to the AXI transfer unit of the DMA, one to the internal control unit of the DMA, and one to the PE control interface of the DMA. From the transfer unit, the retirement buffer receives commands to add a new in-flight transaction at the tail of the queue, to free a successful transaction, or to mark a transaction as failed. In the latter two cases, the buffer is traversed from the head, modifying the first non-free transaction that matches the given AXI ID.

To the DMA control unit, the retirement buffer reports the current number of in-flight and failed bursts and provides the metadata of the next reissuable burst. When at least one burst has failed, the control unit stops issuing new bursts from its queue and waits for all in-flight bursts to complete. Once there are no more in-flight bursts, the control unit reissues bursts as soon as they are reissuable until there are no more failed (and in-flight) bursts. As soon as this is the case, the control unit resumes regular operation by issuing bursts from its queue.

From the PE control interface, the cluster-external address of the first (ordered by request, not response) burst with state ‘failed’ can be read and bursts can be marked as reissuable. For this, a PE reads a DMA register to get the failing external address (or 0 if there is none). Upon such a read, the retirement buffer marks all ‘failed’ bursts with the same page frame number as ‘peeked’ (so that the same page is not reported twice). Meanwhile, the PE determines the missing physical address and adds it to the TLB. When it is done, it writes the handled virtual address to the same DMA register, upon which the retirement buffer marks all ‘failed’ or ‘peeked’ bursts with the same page frame number as ‘reissuable’. Bursts are then reissued by the control unit in the order of their original requests.

V. Results

In this section, we evaluate the performance of our SVM system implemented on an evaluation platform (§ V-A) under various conditions (§ V-B) to demonstrate its significant improvements over the state of the art and identify its limits (§ V-C). Additionally, we discuss how our hybrid-IOMMU-capable DMA engine can save a vast amount of hardware buffers compared to conventional IOMMUs and standard DMA engines (§ V-D).

A. Evaluation Platform

Our evaluation platform is based on the Xilinx Zynq-7045 SoC, which features a dual-core ARM Cortex-A9 CPU, which we use as host processor, and programmable logic, which we use to implement the cluster-based PMCA described in § III. The PEs within a cluster share 8 KiB L1 instruction cache and 256 KiB tightly-coupled L1 data SPM, both split into 16 banks. Ideally, every PE can access one 32-bit word in the L1 SPM per cycle. Every cluster features a multi-channel DMA engine that is parametrized to have up to 8 AXI4 read or write bursts in flight at any time, enabling fast movement of data between L1 and L2 memory or shared DRAM. The PMCA is attached to the host as a memory-mapped device, interfaced through a kernel-level driver and a user-space runtime. The host and the PMCA share 1 GiB of DDR3 DRAM. The hybrid IOMMU features a two-level TLB: The L1 TLB features 32 entries, is fully associative, and translates addresses within a single cycle. The L2 TLB features 256 entries, is 8-way set associative, and translates addresses within up to 6 cycles. The IOMMU connects the PMCA to the Accelerator Coherency Port (ACP) of the Zynq, allowing the PMCA to access the shared main memory coherent to the data caches of the host.

This platform enables us to study and evaluate the system-level integration of a PMCA in a HeSoC. Thus, we did not optimize the PMCA for implementation on the FPGA; the FPGA should be seen as an emulator instead of a fully-optimized 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. Measuring an actual implementation rather than simulating models ensures all aspects and parameters of the evaluated system—including those we did not elaborate in detail in this paper or might have overlooked—are correctly represented in the results.

B. Benchmark Description

To evaluate the performance of our SVM system under various conditions including identifying its limits, we have used two entirely different, configurable benchmark applications. They were obtained by extracting critical phases from real-world applications suitable for implementation on a HeSoC, and by parametrizing them over a large parameter space. They exhibit main-memory access patterns representative for various application domains.

a) Pointer Chasing (PC): This benchmark operates on graphs, stored as vertices linked by pointers. It is representative for wide variety of pointer-chasing applications from the graph processing domain [37]. Prominent examples include breadth-first or shortest path searching, clustering, and PageRank. Due to the irregular and data-dependent access pattern to shared memory and low locality between references, PC represents a worst-case scenario for a virtual memory subsystem. However, SVM is crucial to allow implementations of PC applications at reasonable effort and performance, because offloading a PC application to an accelerator without SVM requires modifying all pointers in a graph. In the benchmark, the host builds up a graph and stores its vertices in a single array in main memory. Every vertex holds the number of successors, a pointer to an array of successor vertex pointers, and a configurable amount of payload data. At the offload, the host passes a pointer to the vertex array and the number of vertices to the PMCA. On the PMCA, all WTs share the work of traversing the vertex array. For each vertex, a WT reads the number of successors and copies the payload data and successor pointers to a buffer in L1 SPM using DMA transfers. The WT then performs a configurable number of computation cycles on the payload and writes the payload to all successors in shared main memory again using DMA transfers.

b) Stream Processing (SP): This benchmark operates on a sequence of data, transferred from and to main memory in regularly strided blocks. It is representative for applications that work on streams of data, and examples range from simple one-dimensional filtering of audio data, over two- and three-dimensional image and video filters, to tensor operations in neural networks. In the benchmark, the host allocates one buffer of configurable size for both input and output (to maximize locality) and then passes the pointer to the buffer and the dimensions of the data blocks to the PMCA. On the PMCA, the WTs share the work of performing a configurable number of computation cycles on each block. Both input and output block are double-buffered in L1 SPM, so that compute and data transfer always overlap.

C. Benchmark Results

In the following plots, we compare the performance of our implementation and the prior state of the art (SoA) to an ideal IOMMU, which translates every address within a single cycle—an unbiased, although practically unreachable baseline. The SoA implementation is from [8], extended to multiple threads on the PMCA. For the relative performance on the y-axis, higher values are better. We evaluate different operational intensities on the x-axis by changing the number of computation cycles per data as described above. The operational intensity of an actual program depends both on the algorithm and the hardware executing it, and this sweep over a range of intensities characterizes our SVM implementation for a given memory access pattern but independent of a very specific program and processing architecture.

a) Pointer Chasing (PC): Fig. 4 shows the performance of PC normalized to an ideal IOMMU over different operational

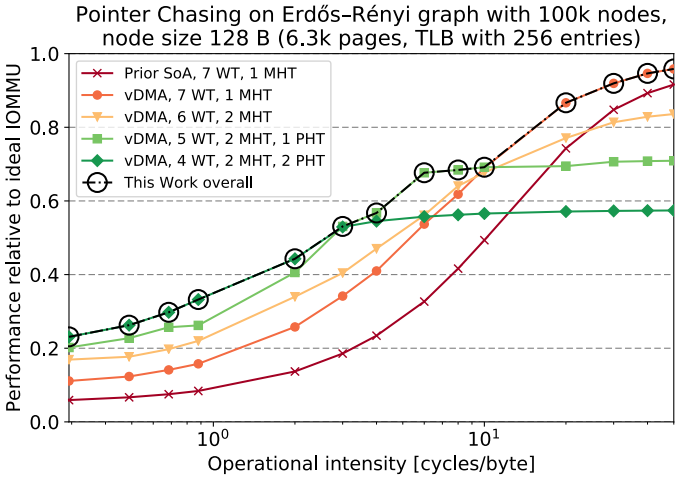

Fig. 4. Pointer Chasing (PC) results for different operational intensities.

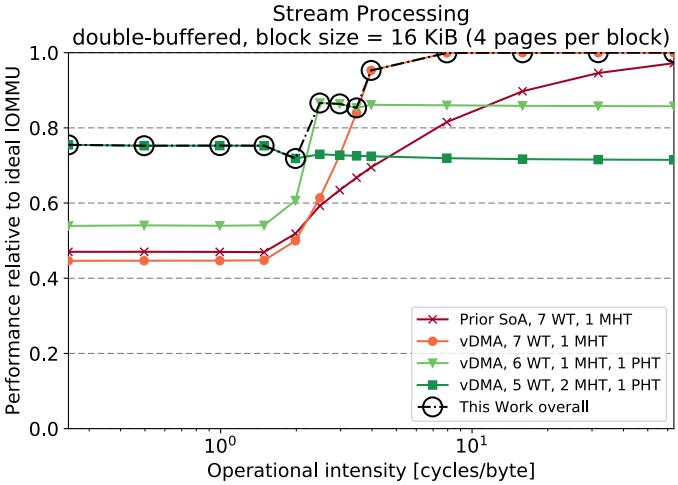

Fig. 5. Stream Processing (SP) results for different operational intensities.

intensities in cycles per byte. For example, a single-precision floating-point implementation of the PageRank algorithm has an operational intensity of 1.2 cycles/B given a floating-point unit (FPU) with a divider and around 10 cycles/B for a reduced-precision fixed-point implementations if no FPU is available.

In the prior SoA, the DMA engine cannot handle TLB misses, so the software must ensure the TLB entries used by a DMA transfer are not replaced while that transfer is running. This locking is the bottleneck of the SoA implementation (first curve in legend order), and limits its performance to less than 50 % below 10 cycles/B. For very high operational intensities, the implementation becomes compute-bound and approaches ideal performance. Our hybrid-IOMMU-compatible DMA engine ('vDMA', all other curves) removes that bottleneck. The second curve is limited by the miss handling throughput of the single MHT for low operational intensities. Replacing one of the WT with another MHT (third curve) resolves this bottleneck. This is effective for low operational intensities, but the missing WT reduces performance in the compute-bound limit. Adding another MHT (not drawn) does not further improve performance because two MHTs are sufficient to handle the misses caused by six WTs. Instead, we replace one of the WTs by a PHT (fourth curve), which causes TLB entries to be set up ahead of the instant the WTs need them. This is very effective in the memory-bound case, increasing performance by another 20 to 30 %. The fourth curve is now prefetch-limited: the single PHT cannot always prefetch early enough, because the PHT itself needs to dereference pointers to determine prefetch targets. Any dereference that causes a TLB miss will block the PHT until the miss is resolved. Thus, replacing another WT with an MHT helps increasing performance in memory-bound cases by an additional 20 %.

Depending on the operational intensity, one of the configurations is optimal. However, as MHTs and PHTs can be inserted in software, e.g., at compile time based on profiling runs or even at run time for largely varying operational intensities, our work significantly improves performance for *all* operational intensities by making optimal use of PEs. The last curve shows the overall optimum configuration. For crucial operational intensities around 1 cycles/B, our work improves performance by 4 x compared to the SoA. For common intensities (arguably below 10 cycles/B), our work improves the SoA performance by at least 50 %.

b) Stream Processing (SP): Fig. 5 shows the performance of SP normalized to an ideal IOMMU over different operational intensities. For example, a one-dimensional FIR filter with N coefficients requires $N/2$ multiply-accumulate operations (MACs) per transferred data word (each word transferred once in, once out), and a matrix-matrix multiplication requires 1 MAC per transferred data word for large matrices. Assuming MACs on the data format are natively supported by the PMCA, the PMCA may compute tens, hundreds, or even thousands of MACs per cycle, depending on the number and architecture of its parallel PEs. Thus, stream processing kernel-architecture combinations may be found anywhere on the x-axis of Fig. 5.

In the prior SoA (first curve in legend order), a WT setting up a DMA transfer ensures that no TLB misses occur during the transfer by explicitly setting up TLB entries and locking them during the transfer. For memory-bound kernels, this is slightly more performant than handling misses by the hybrid-IOMMU-compatible DMA engine (second curve), because the latter stalls on every miss. (If that performance difference was larger, TLB entries could be set up in advance also for the vDMA. In contrast to the prior SoA, where locks on TLB entries had to be coded manually to avoid deadlocks, instructions for setting up TLB entries in advance for the vDMA can be inserted automatically at compile time.) For a range of more compute-intensive kernels, however, this locking is the bottleneck of the SoA implementation, and removing it improves performance by up to 35 %. When only few cycles are executed per transferred byte, performance is dominated by the memory latency in handling TLB misses. Adding another MHT (not drawn) does not change this, because only the input data stream requires one new page at a time. Instead, we replace one of the WTs with a PHT (third curve), which increases performance by 20 to 40 % in the memory-bound case. This configuration is limited by the *throughput* of the MHT, and because the PHT causes more than one page to be outstanding in the miss queue, there is now work

for another MHT. Indeed, adding another MHT (fourth curve) increases performance by another 40 %, up to the point where it limited by the throughput of the PHT in the memory-bound case and by the five WTs in the compute-bound case. Adding another PHT might increase performance even further, but the current PHT generation algorithm does not support distributing the prefetches for a single memory access stream among two PHTs.

The last curve shows the overall optimum of all configurations of our work. Our work improves performance compared to the SoA by up to 60 % for memory-intense kernels, and reduces the overhead compared to an ideal IOMMU to below 25 % for *any* operational intensity. Our work also reduces the operational intensity at which that overhead is below 10 % to ca. 4 cycles/B. As the optimal configuration again can be selected at compile time or even changed at run time, our work significantly improves performance over the full spectrum of operational intensities also for very regular memory access patterns by replacing WTs with MHTs or PHTs when it improves overall performance.

D. Hardware Requirements of Hybrid-IOMMU-Capable DMA

Making the DMA engine compatible with the hybrid IOMMU not only improves performance compared to the SoA, it also dramatically reduces the amount of memory required to buffer DMA bursts that miss in the TLB. Our DMA engine is parametrized to have up to 8 AXI 4 read or write bursts in flight at any time. Each burst can transfer up to 2 KiB. The total maximum amount of data in flight is 16 KiB, and a buffer of this size would be required to enable other masters to continue accessing SVM in the worst case scenario where 8 write bursts miss in the TLB. The retirement buffer in our DMA engine stores just the metadata of each burst: 32 bit for the virtual start address, 16 bit for the local start address, 3 bit for the ID, 8 bit for the length of the burst, and 3 bit for the status of the burst; less than 8 B in total. Thus, the retirement buffer requires just 64 B for the same TLB miss tolerance as the 16 KiB data buffer—a factor 256 less.

VI. Conclusion

In this work, we presented and evaluated our scalable and efficient SVM solution for HeSoCs. It is based on a hybrid IOMMU and advances the state of the art in three important ways: First, compiler-generated PHTs proactively fill the TLB to minimize the rate of TLB misses. Second, a variable number of parallel PHTs handle TLB misses to scale the miss handling throughput with the demand. Third, a hybrid-IOMMU-capable DMA engine supports parallel burst DMA transfers to SVM without additional buffers. Compared to the state of the art, our work improves PMCA performance for memory-intensive kernels by up to 4× for irregular and by up to 60 % for regular memory access patterns. Compared to using data buffers to absorb bursts from DMA engines in a conventional IOMMU, our solution requires two orders of magnitude less memory and scales better, as it only stalls the missing DMA engine. In the future, we plan to explore compiler-generated PHTs for kernels that mandate speculative prefetching, improve per-thread miss handling throughput by supporting out-of-order page table walking, and avoid stalling the entire DMA on a TLB miss while maintaining memory order guarantees.

Acknowledgments

The authors thank J. Weinbuch for his work on multi-threaded TLB miss handling during his Master's Thesis.

References

[1] HSA Foundation, "HSA Foundation," 2012, www.hsafoundation.com.
[2] J. Stuecheli *et al.*, "CAPI: A coherent accelerator processor interface," *IBM Journal of R&D*, vol. 59, Jan 2015.
[3] J. Goodacre, "The evolution of the ARM architecture towards big data and the data-centre," in *VHPC '13*.
[4] B. Pichai *et al.*, "Architectural support for address translation on GPUs: Designing memory management units for CPU/GPUs with unified address spaces," in *ASPLOS '14*.
[5] ARM Ltd., "ARM CoreLink MMU-500 system memory management unit," Technical reference manual, 2016.
[6] M. Lavasani *et al.*, "An FPGA-based in-line accelerator for memcached," *IEEE CAL*, vol. 13, 2014.
[7] P. Vogel *et al.*, "Lightweight virtual memory support for zero-copy sharing of pointer-rich data structures in heterogeneous embedded SoCs," *IEEE TPDS*, vol. 28, 2017.
[8] P. Vogel *et al.*, "Efficient virtual memory sharing via on-accelerator page table walking in heterogeneous embedded SoCs," *ACM TECS*, vol. 16, Sep. 2017.
[9] J. Vesely *et al.*, "Observations and opportunities in architecting shared virtual memory for heterogeneous systems," in *IEEE ISPASS '16*.
[10] A. Roth *et al.*, "Effective jump-pointer prefetching for linked data structures," in *ISCA '99*.
[11] H. Al-Sukhni *et al.*, "Compiler-directed content-aware prefetching for dynamic data structures," in *IEEE PACT '03*.
[12] Intel Corp., "The compute architecture of Intel Processor Graphics Gen9," 2015.
[13] G. Kornaros *et al.*, "I/O virtualization utilizing an efficient hardware system-level memory management unit," in *ISSoC '14*.
[14] Xilinx Inc., "Zynq UltraScale+ MPSoC data sheet: Overview," Advance Product Specification, 2017.
[15] J. Power *et al.*, "Supporting x86-64 address translation for 100s of GPU lanes," in *IEEE HPCA '14*.
[16] PCI-SIG, "PCIe Address Translation Services (ATS)," Standard Spec., Jan. 2009.
[17] C.-K. Luk *et al.*, "Automatic compiler-inserted prefetching for pointer-based applications," *IEEE TC*, vol. 48, Feb 1999.
[18] M. Karlsson *et al.*, "A prefetching technique for irregular accesses to linked data structures," in *IEEE HPCA '00*.
[19] I. Ganusov *et al.*, "Efficient emulation of hardware prefetchers via event-driven helper threading," in *IEEE PACT '06*.
[20] E. Ebrahimi *et al.*, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *IEEE HPCA '09*.
[21] J. Lee *et al.*, "Prefetching with helper threads for loosely coupled multiprocessor systems," *IEEE TPDS*, vol. 20, Sept 2009.
[22] S. W. Son *et al.*, "A compiler-directed data prefetching scheme for chip multiprocessors," in *PPoPP '09*.
[23] R. Cooksey *et al.*, "A stateless, content-directed data prefetching mechanism," in *ASPLOS '02*.
[24] J. Collins *et al.*, "Pointer cache assisted prefetching," in *ACM/IEEE MICRO '02*.
[25] C.-K. Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *ISCA '01*.
[26] A. Saulsbury *et al.*, "Recency-based TLB preloading," in *ISCA '00*.
[27] G. B. Kandiraju *et al.*, "Going the distance for TLB prefetching: An application-driven study," in *ISCA '02*.
[28] D. Lustig *et al.*, "TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs," *ACM TACO*, vol. 10, Apr. 2013.
[29] D. Melpignano *et al.*, "Platform 2012, a many-core computing accelerator for embedded SoCs," in *ACM/IEEE DAC '12*.
[30] Kalray S.A., "MPPA MANYCORE," 2014.
[31] D. Rossi *et al.*, "Energy-efficient near-threshold parallel computing: The PULPv2 cluster," *IEEE Micro*, vol. 37, Sept 2017.
[32] I. Loi *et al.*, "The quest for energy-efficient I$ design in ultra-low-power clustered many-cores," *IEEE TMSCS*, vol. 4, Apr 2018.
[33] I. Loi *et al.*, "Exploring multi-banked shared-L1 program cache on ultra-low power, tightly coupled processor clusters," in *ACM CF '15*.
[34] D. Rossi *et al.*, "Ultra-low-latency lightweight DMA for tightly coupled multi-core clusters," in *ACM CF '14*.
[35] T. Sherwood *et al.*, "Discovering and exploiting program phases," *IEEE Micro*, vol. 23, Nov 2003.
[36] A. Marongiu *et al.*, "Simplifying many-core-based heterogeneous SoC programming with offload directives," *IEEE TII*, vol. 11, Aug 2015.
[37] Y. Guo *et al.*, "How well do graph-processing platforms perform? An empirical performance evaluation and analysis," in *IEEE IPDPS '14*.