Supporting Address Translation for Accelerator-Centric Architectures

ABSTRACT

While emerging accelerator-centric architectures offer orders-of-magnitude performance and energy improvements, use cases and adoption can be limited by their rigid programming model. A unified virtual address space between the host CPU cores and customized accelerators can largely improve the programmability, which necessitates hardware support for address translation. However, supporting address translation for customized accelerators with low overhead is nontrivial. Prior studies either assume an infinite-sized TLB and zero page walk latency, or rely on a slow IOMMU for correctness and safety which penalizes the overall system performance.

To provide efficient address translation support for accelerator-centric architectures, we examine the memory access behavior of customized accelerators to drive the TLB augmentation and MMU designs. First, to support bulk transfers of consecutive data between the scratchpad memory of customized accelerators and the memory system, we present a relatively small private TLB design to provide low-latency caching of translations to each accelerator. Second, to compensate the effects of the widely used *data tiling* techniques, we design a shared level-two TLB to serve private TLB misses on common virtual pages, eliminating duplicate page walks from accelerators working on neighboring data tiles that are mapped to the same physical page. This two-level TLB design effectively reduces page walks by 75.8% on average. Finally, instead of implementing a dedicated MMU which introduces additional hardware complexity, we propose simply leveraging the host per-core MMU for efficient page walk handling. This mechanism is based on our insight that the existing MMU cache in the CPU MMU satisfies the demand of customized accelerators with minimal overhead. Our evaluation demonstrates that the combined approach incurs only 6.4% performance overhead compared to the ideal address translation.

1. INTRODUCTION

In light of the failure of Dennard scaling and recent slowdown of Moore's law, the computer architecture community has proposed many heterogeneous systems that combine conventional processors with a rich set of customized accelerators onto the same die [18, 19, 22, 30, 31, 36, 39, 43, 46, 47, 49, 50, 52]. Such accelerator-centric architectures trade dark, unpowered silicon [21, 54] area for customized accelerators that offer orders-of-magnitude performance and energy gains compared to general-purpose cores. These accelerators are usually application-specific implementations of a particular functionality, and can range from simple tasks (e.g., a multiply-accumulate operation) to complex applications (e.g., medical imaging [20], database management [35], Memcached [37, 38]).

While such architectures promise tremendous performance/watt targets, system architects face a multitude of new problems, including but not limited to i) how to integrate customized accelerators into the existing memory hierarchies and operating systems, ii) how to efficiently offload algorithm kernels from general-purpose cores to customized accelerators. One of the key challenges involved is the memory management between the host CPU cores and accelerators. For conventional physically addressed accelerators, if the application lives in the user space, an offload process requires copying data across different privilege levels to/from the accelerator and manually maintaining data consistency. Additional overhead in data replication and OS intervention is inevitable, which may diminish the gain of customization [14]. Zero-copy avoids copying buffers via operating system support. However, programming with special APIs and carefully managing buffers can be a giant pain for developers.

While accelerators in current heterogeneous systems have limited support for virtual addresses, industry initiatives, such as the Heterogeneous System Architecture (HSA) foundation, are proposing to shift towards a unified virtual address between the host CPU and accelerators [25]. In this model, instead of maintaining two copies of data in both host and device address spaces, only a single allocation is necessary. As a consequence, an offload process simply requires passing the virtual pointer to the shared data to/from the accelerator. This has a variety of benefits, including the elimination of explicit data copying, increased performance of fine-grained memory accesses, the support for cache coherence and memory protection.

Unfortunately, the benefits of unified virtual address also come at a cost. A key requirement of virtually addressed accelerators is the hardware support for virtual-

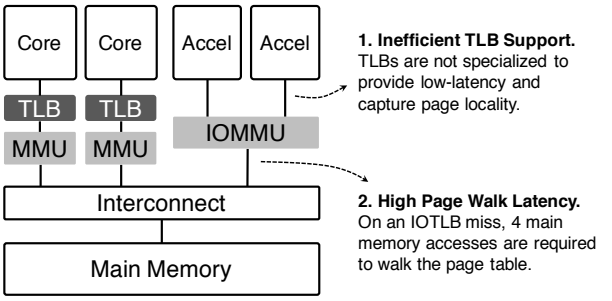

Figure 1: Problems in current address translation support for accelerator-centric architectures in an IOMMU-only configuration

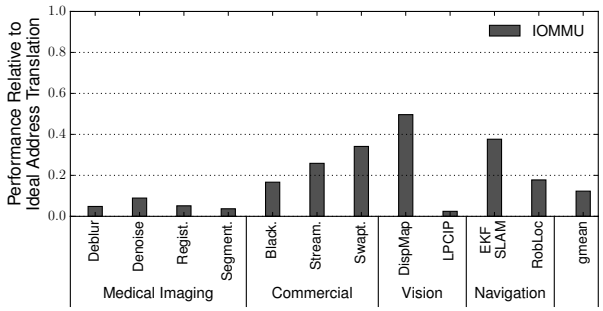

Figure 2: Performance of the baseline IOMMU approach relative to ideal address translation

to-physical address translation. Commercial CPUs and SoCs have introduced I/O memory management units (IOMMUs) [2, 5, 28, 34] to allow loosely-coupled devices to handle virtual addresses, as is shown in Figure 1. These IOMMUs have I/O translation lookaside buffers (IOTLBs) and logic to walk the page table, which can provide address translation support for customized accelerators. However, an naive IOMMU configuration cannot meet the requirement of today's high-performance customized accelerators as it lacks efficient TLB support and excessively long latency is incurred to walk the page table on IOTLB misses. Figure 2 shows that the performance of the baseline IOMMU approach achieves only 12.3% of the ideal address translation where all translation requests hit in an ideal TLB[1], leaving a huge performance gap for improvement. Recent advances in IOMMU enable translation caching in devices [34]. However, designing efficient TLBs for high-performance accelerators is non-trivial and should be carefully studied. Prototypes in prior studies encounter the challenge of virtual address support [15, 37, 38, 42] as well. However, their focus is mainly on the design and performance of accelerators, with either the underlying address translation approach not detailed or the performance impact not evaluated.

In this paper, **our goal** is to provide an efficient address translation support for heterogeneous customized accelerator-centric architectures. The hope is that such design can enable a unified virtual address space between host CPU cores and accelerators with modest hardware modification and low performance overhead compared to the ideal address translation.

By examining the memory access behavior of customized accelerators, we propose an efficient hardware support for address translation tailored to the specific challenges and opportunities of accelerator-centric architectures that includes:

1. **Private TLBs.** Unlike conventional CPUs and GPUs, customized accelerators typically exhibit bulk transfers of consecutive data when loading data into the scratchpad memory and writing data back to the memory system. Therefore, a relatively small (16-32 entries) and low-latency private TLB can not only allow accelerators to save trips to IOMMU, but also capture the page access locality. On average, a private TLB with 32 entries can reduce 30.4% of the page walks compared to the IOMMU-only baseline, and improves the performance from 12.3% (IOMMU baseline) to 23.3% of the ideal address translation.
2. **A Shared TLB.** *Data tiling* techniques are widely used in customized accelerators to improve the data reuse within each tile and the parallelism between tiles. Due to capacity limit of each accelerator's scratchpad memory, this usually breaks the contiguous memory region within a physical page into multiple data tiles that are mapped to different accelerator instances for parallelism. In light of this, we present a shared level-two TLB design to filter translation requests on common pages so that duplicate page walks will not be triggered from accelerator instances working on neighboring data tiles. Our evaluation shows that a two-level TLB design with a 512-entry shared TLB can reduce page walks by 75.8% on average, and improves the performance to 51.8% of the ideal address translation.
3. **Host Page Walks.** As accelerators are sensitive to long memory latency, the excessively long latency of page walks that cannot be filtered by TLBs degrades the system performance. While enhancing the IOMMU with MMU caches or introducing a dedicated MMU for accelerators are viable approaches, better opportunities lie in the coordination between the host core and the accelerators invoked by it. The idea is that by extending the per-core MMU to provide an interface, accelerators operating within the same application's virtual address space can offload TLB misses to the page walker of the host core MMU. The benefits come in three ways: first, the page walk latency can be significantly reduced due to the presence of MMU caches [6, 8, 27] in the host core MMU; second, prefetching effects can be achieved due to the support of data cache as loading one cacheline effectively brings in multiple page table entries; third, cold misses in the MMU cache and data cache can be minimized since it is likely that the host core has already touched the data structure before offloading so that corresponding resources have been warmed-up. The experimental results show the host page walk reduces the average page walk latency to 58 cycles across different benchmarks, and the combined

[1]Detailed experimental setup is described in Section 3. More analysis of this gap is presented in Section 4.1.

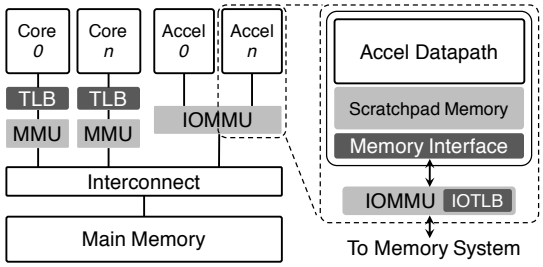

Figure 3: A detailed look at an accelerator connecting with system IOMMU (baseline)

approach bridges the performance gap to 6.4% compared to the ideal address translation.

The rest of this paper is organized as follows. Section 2 characterizes address translation behaviors of customized accelerators to motivate our design. Section 3 explains our simulation methodology and workloads. Section 4 details the design and evaluation of the proposed architectural support for address translation. Section 5 discusses more use cases. Section 6 summarizes related work and Section 7 concludes the paper.

2. CHARACTERIZATION OF CUSTOMIZED ACCELERATORS

2.1 Accelerator-Centric Architectures

We present an overview of the baseline accelerator-centric architecture used throughout this paper in Figure 1. In this architecture, CPU cores and loosely-coupled accelerators share the physical memory. Each CPU core has its own TLB and MMU, while all accelerators share an IOMMU that has an IOTLB inside it. A CPU core can launch one or more accelerators by offloading a task to them for superior performance and energy efficiency. Launching multiple accelerators can exploit data parallelism by assigning accelerators to different data partitions, which we call *tiles*.

The details of a customized accelerator are shown in Figure 3. In contrast to general-purpose CPU cores or GPUs, accelerators do not use instructions and feature customized registers and datapaths with deep pipelines [46]. Scratchpad memory (SPM) is predominantly used by customized accelerators instead of hardware-managed caches, and data layout optimization techniques such as *data tiling* is often applied for increased performance. A memory interface such as a DMA (direct memory access) is often used to transfer data between the SPM and the memory system.

Due to these microarchitectural differences, customized accelerators exhibit distinct memory access behaviors compared to CPUs and GPUs. To drive our design, we characterize such behaviors in the following subsections: the bulk transfer of consecutive data, the impact of data tiling, and the sensitivity to address translation latency.

2.2 Bulk Transfer of Consecutive Data

The performance and energy gains of customized accelerators are largely due to the removal of instructions through specialization and deep pipelining [46]. To guarantee a high throughput for such customized pipelines—processing one input data every II (pipeline initialization interval) cycles, where II is usually one or two—the entire input data must be available in the SPM to provide register-like accessibility. Therefore, the execution process of customized accelerators typically has three phases: reading data from the memory system to the SPM in bulk for local handling, pipelined processing on local data, and then writing output data back to the memory system. Such bulky reads and writes appear as multiple streams of consecutive accesses in the memory system, which exhibit good memory page locality and high memory bandwidth utilization.

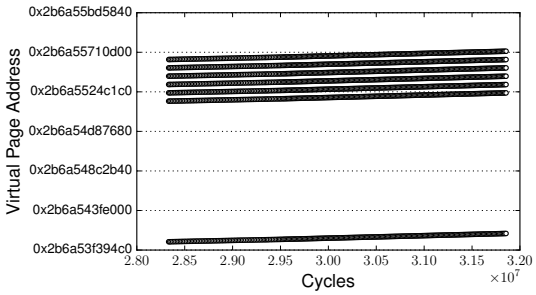

Figure 4: TLB miss behavior of BlackScholes

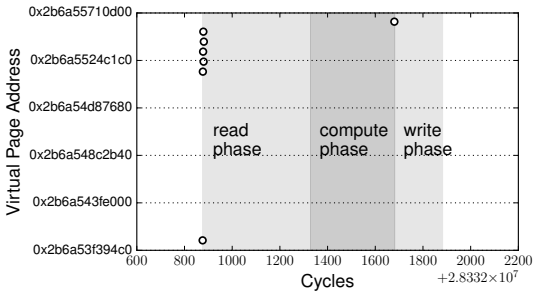

Figure 5: TLB miss trace of a single execution from BlackScholes

To demonstrate such characteristics, we plot the the trace of virtual pages that trigger TLB misses in BlackScholes in Figure 4 (our simulation methodology and workloads are detailed in Section 3). We can see that the TLB miss behavior is extremely regular, which is different from the more random accesses in CPU or GPU applications. As accelerators feature customized deep pipeline without multithreading or context switching, the page divergence is only determined by the number of input data arrays and the dimentionality of each array. Figure 5 confirms this by showing the TLB miss trace in a single execution of BlackScholes, which accesses six one-dimensional input arrays and one output array. In addition, we can see that TLB misses typically happen at the beginning of the bulky data read and write phases, followed by large number of TLB hits. Therefore, high hit rates can be expected from TLBs with sufficient capacity.

This type of regularity is also observed for a string-matching application and is reported to be common for a wide range of applications such as image processing and graphics [51]. We think that this characteris-

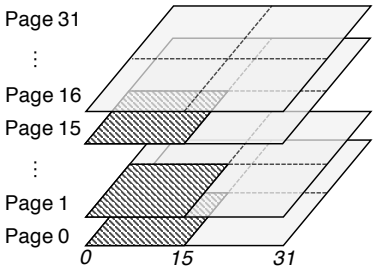

Figure 6: Rectangular tiling on a $32 \times 32 \times 32$ data array into $16 \times 16 \times 16$ tiles. Each tile accesses 16 pages and can be mapped to a different accelerator for parallel processing.

tic is determined by the fundamental microarchitecture rather than the application domain. Such regular access behavior presents opportunities for relatively simple designs in supporting address translation for accelerator-centric architectures.

2.3 Impact of Data Tiling

Data tiling techniques are widely used on customized accelerators, which groups data points into tiles that are executed atomically. As a consequence, each data tile can be mapped to a different accelerator to maximize the parallelism. Also, data tiling can improve data locality for the accelerator pipeline, leading to an increased computation to communication ratio. This also enables the use of double (ping-pong) buffering.

While the input data array could span several memory pages, the tile size of each input data is usually smaller than a memory page due to limited SPM resources, especially for high-dimensional arrays. As a result, neighboring tiles are likely to be in the same memory page. These tiles, once mapped to different accelerators, will trigger multiple address translation requests on the same virtual page. Figure 6 shows a simple example of tiling on a $32 \times 32 \times 32$ input float array with $16 \times 16 \times 16$ tile size, producing 8 tiles in total. This example is derived from the medical imaging applications, while the sizes are picked for illustration purposes only. As the first two dimensions can be exactly fit into a 4KB page, 32 pages in total are allocated for the input data. Processing one tile needs to access 16 of the 32 pages. However, mapping each tile to a different accelerator will trigger $16 \times 8 = 128$ address translation requests, which is 4 times more than the minimum 32 requests. Such duplication in address translation requests must be resolved so that additional translation service latency can be avoided. The simple coalescing logic used in GPUs would not be sufficient because concurrently running accelerators are not designed to execute in lockstep.

2.4 Address Translation Latency Sensitivity

While CPUs expose memory-level parallelism (MLP) using large instruction windows and GPUs leverage their extensive multithreading to issue bursts of memory references, accelerators generally lack architectural support for fine-grained latency hiding. As discussed

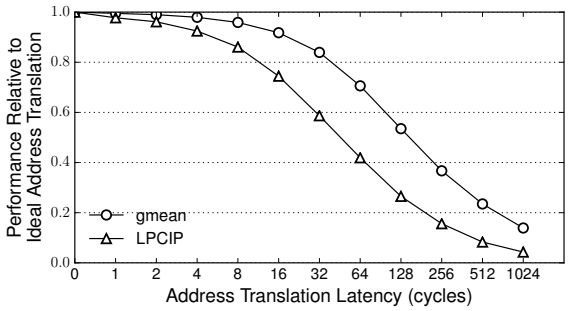

Figure 7: Geometric mean slowdown over all benchmarks with varied address translation latencies, with LPCIP being the most sensitive benchmark

earlier, the performance of customized accelerators relies on predictable accesses to the local SPM. Therefore, the computation pipeline cannot start until the entire input data tile is ready. To alleviate this problem, double buffering techniques are commonly used to overlap communication with computation: processing on one buffer while transferring data on the other. However, such coarse-grained techniques require a careful design to balance communication and computation, and can be ineffective in tolerating long-latency memory operations, especially page walks on TLB misses.

To further demonstrate latency sensitivity, we run simulations with varied address translation latencies added to each memory reference. Figure 7 presents the performance slowdown of LPCIP and the geometric mean slowdown over all benchmarks from additional latency. In general, address translation latency within 8 cycles can be tolerated by double buffering. Any additional latency beyond 16 cycles significantly degrades overall system performance. LPCIP shows the highest sensitivity to additional cycles among all benchmarks since the accelerator issues dynamic memory accesses during the pipelined processing which is beyond the coverage of double buffering. While GPUs are reported to be able to tolerate 600 additional memory access cycles with a maximum slowdown of only 5% [24], the performance of accelerators will be decreased by 5x with the same additional cycles.

Such immense sensitivity poses serious challenges to designing an efficient address translation support for accelerators: (1) TLBs must be carefully designed to provide low access latency, (2) as page walks incur long latency which could be a few hundred cycles, TLB structures must be effective in reducing the number of page walks, (3) for page walks that cannot be avoided, page walker must be optimized for lower latency.

3. SIMULATION METHODOLOGY

Simulation. We use PARADE [17], an open-source cycle-accurate full-system simulator to evaluate the accelerator-centric architecture. PARADE extends the gem5 [12] simulator with high-level synthesis support [16] to accurately model the accelerator module, including the customized data path, the associated SPM and the DMA interface.

We model an 8-issue out-of-order X86-64 CPU core

Table 1: Benchmark descriptions with input sizes and number of heterogeneous accelerators. [20]

Domain	Application	Algorithmic Functionality	Input Size	Accel[2] Types
Medical Imaging	Deblur	Total variation minimization and deconvolution	128 slices of images, each image of size 128×128	4
	Denoise	Total variation minimization		2
	Registration	Linear algebra and optimizations		2
	Segmentation	Dense linear algebra, and spectral methods		1
Commercial from Parsec [11]	BlackScholes	Stock option price prediction using floating point math	256K sets of option data	1
	StreamCluster	Clustering and vector arithmetic	64K 32-dimensional streams	5
	Swaptions	Computing swaption prices by Monte Carlo simulation	8K sets of option data	4
Computer Vision	Disparity Map	Calculate sums of absolute differences and integral image representations using vector arithmetic	Images pairs of size 64×64 8×8 window, 64 max. disparity	4
	LPCIP Desc	Log-polar forward transformation of image patch around each feature	128K features from images of size 640×480	1
Computer Navigation	EKF SLAM	Partial derivative, covariance, and spherical coordinate computations	128K sets of sensor data	2
	Robot Localization	Monte Carlo Localization using probabilistic model and particle filter	128K sets of sensor data	1

at 2GHz with 32KB L1 instruction and data cache, 2MB L2 cache and a per-core MMU. We implement a wide spectrum of accelerators, as shown in Table 1, where each accelerator can issue 64 outstanding memory requests and has double buffering support to overlap communication with computation. The host core and accelerators share 2GB DDR3 DRAM on four memory channels. We extend PARADE to model an IOMMU with a 32-entry IOTLB [3]. To study the overhead of address translation, we model an ideal address translation in the simulator with infinite-sized TLBs and zero page walk latency for accelerators. We assume 4KB page size is used in the system for the best compatibility. The impact of using large pages will be discussed in Section 5.2. Table 2 summarizes the major parameters used in our simulation.

Table 2: Parameters of the baseline architecture

Component	Parameters
CPU	1 X86-64 OoO core @ 2GHz 8-wide issue, 32KB L1, 2MB L2
Accelerator	4 instances of each accelerator 64 outstanding memory references Double buffering enabled
IOMMU	4KB page, 32-entry IOTLB
DRAM	2GB, 4 channels, DDR3-1600

Workloads. To provide a quantitative evaluation of our address translation proposal, we use a wide range of applications that are open-sourced together with PARADE. These applications can be categorized into four domains: medical imaging, computer vision, computer navigation and commercial benchmarks from PARSEC [11]. A brief description of each application, its input size and the number of different accelerator types involved is specified in Table 1. Each application may call one or more types of accelerators to perform different functionalities corresponding to the algorithm in various phases. In total, we implement 25 types of accelerators [2]. To achieve maximum performance, multiple instances of the same type can be invoked by the host to process in parallel. By default, we use four instances of each type in our evaluation unless otherwise specified. There will be no more than 20 active accelerators while others will be powered off according to the application that is running.

[2]The racian accelerator is shared by Deblur and Denoise; the gaussian accelerator is shared by Deblur and Registration.

4. DESIGN AND EVALUATION OF ADDRESS TRANSLATION SUPPORT

The goal of this paper is to design an efficient address translation support for accelerator-centric architectures. After carefully examining the distinct memory access behaviors of customized accelerators in Section 2, we propose the corresponding TLB and MMU designs with quantitative evaluations step by step.

4.1 Gap between the Baseline IOMMU Approach and the Ideal Address Translation

Figure 2 shows the performance of the baseline IOMMU approach relative to the ideal address translation with infinite-sized TLBs and zero page walk latency. As an IOMMU-only configuration requires each memory reference to be translated by the centralized hardware interface, performance suffers from frequent trips to the IOMMU. On one hand, benchmarks with large page reuse distances, such as the medical imaging applications, experience IOTLB thrashing due to the limited capacity. In such cases, IOTLB cannot provide effective translation caching, leading to a large number of long-latency page walks. On the other hand, while the IOTLB may satisfy the demand of some benchmarks with small page reuse distances, such as computer navigation applications, the IOMMU lacks efficient page walk handling, which significantly degrades system performance on IOTLB misses. As a result, the IOMMU approach achieves only an average of 12.3% relative to the performance of the ideal address translation, leaving a huge performance gap.

In order to bridge the performance gap, we propose to reduce the address translation overhead in three steps: (1) providing low-latency access to translation caching by allowing customized accelerators to store physical addresses locally in TLBs, (2) reducing the number of page walks by exploiting page sharing between accelerators resulted from data tiling, and (3) minimizing the

page walk latency by offloading the page walk request to the host core MMU. We detail our designs and evaluations in the following subsections.

4.2 Private TLBs

To enable more capable devices such as accelerators, recent IOMMU proposals allow IO devices to cache address translation in devices [34]. This reduces the address translation latency on TLB hits and relies on the page walker in IOMMU on TLB misses. However, the performance impact and design tradeoffs are not scrutinized in the literature.

4.2.1 Implementation

TLB sizes. While a large TLB may have higher hit rate, smaller TLB sizes are preferable in providing lower access latency, since customized accelerators are very sensitive to the address translation latency. Moreover, TLBs are reported to be power-hungry and even TLB hits consume a significant amount of dynamic energy [33]. Thus TLB sizes must be carefully chosen.

Commercial CPUs currently implement 64-entry per-core L1 TLBs and recent GPU studies [44, 45] introduce 64-128 entry post-coalescer TLBs. As illustrated in Section 2.2, customized accelerators have much more regular and less divergent access patterns compared to general-purpose CPUs and GPUs, which advocates for a relatively small private TLB size for shorter access latency and lower energy consumption. Next we quantitatively evaluate the performance effects of various private TLB sizes. We assume a least-recently-used (LRU) replacement policy to capture locality.

Private TLB size for all benchmarks except medical imaging applications. Figure 8 illustrates that all seven evaluated benchmarks greatly benefit from adding private TLBs. In general, small TLB sizes such as 16-32 entries suffice to achieve most of the improved performance. The cause of this gain is twofold: (1) accelerators benefit from reduced access time to locally cached translations; (2) even though the capacity is not enlarged compared to the 32-entry IOTLB, accelerators enjoy private TLB resource rather than sharing the IOTLB. LPCIP receives the largest performance improvement from having a private TLB. This matches the observation that it has the highest sensitivity to address translation latency due to dynamic memory references during pipelined processing, since providing a low-latency private TLB greatly reduces pipeline stalls.

Private TLB size for medical imaging applications. Figure 9 shows the evaluation on the four medical imaging benchmarks. These benchmarks have larger memory footprint with more input array references, and the three-dimensional access pattern (accessing multiple pages per array reference as demonstrated in Figure 6) further stresses the capacity of private TLBs. While apparently 256-entry achieves the best performance for the four, the increased TLB access time would decease performance for other benchmarks, especially latency-sensitive ones such as LPCIP. In addition, a large TLB will also consume more energy.

Non-blocking design. Most CPUs and GPUs use

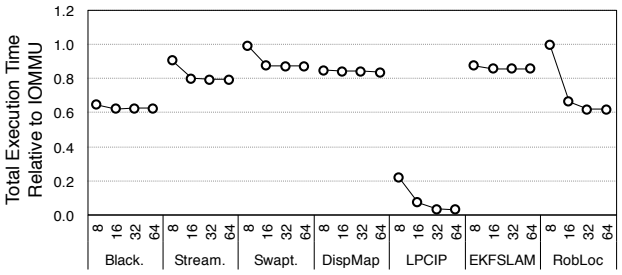

Figure 8: Performance for benchmarks other than medical imaging with various private TLB sizes, assuming fixed access latency.

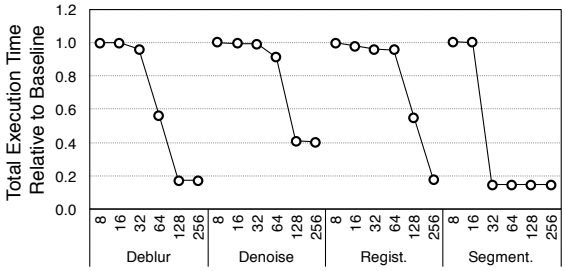

Figure 9: Performance for medical imaging benchmarks with various private TLB sizes, assuming fixed access latency.

blocking TLBs since the latency can be hidden with wide MLP. In contrast, accelerators are sensitive to long TLB miss latencies. Blocking accesses on TLB misses will stall the data transfer, reducing the memory bandwidth utilization which results in performance degradation. In light of this, our design provides non-blocking hit-under-miss support to overlap TLB miss with hits to other entries.

Correctness issues. In practice, correctness issues including page faults and TLB shootdowns [13] have negligible effects on the experimental results. We discuss them here for implementation purposes. While page faults can be handled by the IOMMU, accelerator private TLBs must support TLB shootdowns from the system. In a multi-core-accelerator system, if the mapping of memory pages are changed, all sharers of the virtual memory are notified to invalidate TLBs using TLB shootdown inter-processor interrupts (IPIs). We assume shootdowns are supported between CPU TLBs and the IOTLB based on this approach. We also extend IOMMU to send invalidation to accelerator private TLBs to flush stale values.

4.2.2 Evaluation

We find that low TLB access latency provided by local translation caching is key to the performance of customized accelerators. While the optimal TLB size appears to be application-specific, we choose 32-entry for balance between latency and capacity for our benchmarks, and also lower energy consumption. On average, the 32-entry private TLB achieves 23.3% of the ideal address translation performance, one step up from the 12.3% IOMMU baseline. Further improvements are possible by customizing TLB sizes and supporting miss-

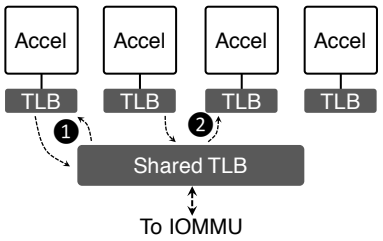

Figure 10: The structure of a shared level-two TLB

under-miss targeting individual applications. We leave these for future work.

4.3 A Shared Level-Two TLB

Figure 10 depicts the basic structure of our two-level TLB design, and illustrates two orthogonal benefits provided from adding a shared TLB. First, the requested entry is previously inserted to the shared TLB by a request from the same accelerator. This is the case when the private TLB size is not sufficient to capture the reuse distance so that the requested entry is evicted from the private TLB earlier. Specifically, medical imaging benchmarks would benefit from having a large shared TLB. Second, the requested entry is previously inserted to the shared TLB by a request from another accelerator. This case is common when data tile size is smaller than a memory page and neighboring tiles within a memory page are mapped to different accelerators (illustrated in Section 2.3). Once an entry is brought into the shared TLB by one accelerator, it is immediately available to other accelerators, leading to shared TLB hits. Requests that also miss in the shared TLB need to access IOMMU for page table walking.

4.3.1 Implementation

TLB size. While our design trades capacity for lower access latency in private TLBs, we provide relatively larger capacity in the shared TLB to avoid thrashing. Based on the evaluation of performance impact of private TLB sizes, we assume a 512-entry shared TLB for the four-accelerator-instances case, where LRU replacement policy is used. Though it virtually provides a 128-entry level-two TLB for each sharer, a shared TLB is more flexible in allocating resources to a specific sharer, resulting in improved performance.

Non-blocking design. Similar to the private TLBs, our shared TLB design also provides non-blocking hit-under-miss support to overlap TLB miss with hit accesses to other entries.

Inclusion policy. In order to reap the benefits of aforementioned use cases, entries requested by an accelerator must be inserted in both the private and the shared TLB. We adopt the approach in [10] to implement a *mostly-inclusive* policy, where each TLB is allowed to make independent replacement decisions. This relaxes the coordination between private and shared TLBs and simplifies the control logic.

Placement. We provide a centralized shared level-two TLB not tied to any of the accelerators. This requires each accelerator to send requests through the interconnect to access the shared TLB which adds additional latency. However, we find the benefit completely outweighs the added access latency for the current configuration. Much larger TLB sizes or more sharers (we will discuss this in Section 5.1) could benefit from a banked design, but such use cases are not well established.

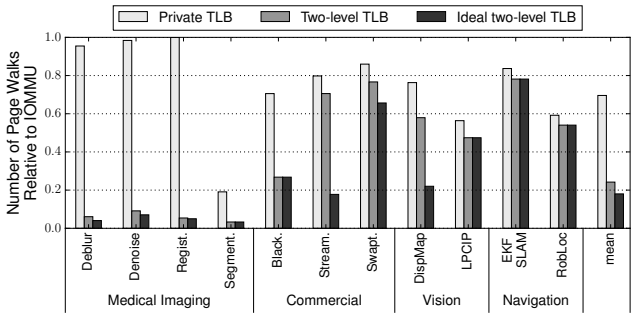

Figure 11: Page walk reduction compared to the IOMMU baseline[3]

Correctness issues. In addition to the TLB shootdown support in private TLBs, the shared TLB also needs to be checked for invalidation. The reason is in a mostly-inclusive policy, entries that are previously brought in can be present in both levels.

4.3.2 Evaluation

In contrast to private TLBs where low access latency is the key, the shared TLB mainly aims to reduce the number of page walks in two ways: (1) providing a larger capacity to capture the page locality for applications which is difficult to achieve in private TLBs without sacrificing access latency, (2) reducing TLB misses on common virtual pages by enabling translation sharing between concurrent accelerators.

Figure 11 sheds light upon the page walk reduction[3] in our two-level TLB design. Compared to private TLBs only, adding a shared TLB consistently reduces the overall number of page walk requests. For medical imaging benchmarks, especially Deblur, Denoise and Registration, which suffer from insufficient private TLB capacity, the shared TLB significantly cuts the number of page walks by providing more resource. For benchmarks that already find enough entries in private TLBs, such as Segmentation and BlackScholes, the shared TLB reduces the number of page walks by decreasing TLB misses on common virtual pages, which is due to data tiling effects. In general, the two-level TLB design achieves 76.8% page walk reduction compared to the IOMMU approach, leaving only a small gap to an ideal two-level TLB. StreamCluster and DisparityMap involve multiple iterations over the same input data using different types of accelerators. The result shows a gap between the 512-entry case and the ideal case because the size of input data exceeds the reach of the 512-entry TLB, but has no problem fitting in the infinite-sized TLB which eliminates cold misses at the beginning of each iteration.

[3]Note that the number of page walks does not equal to the number of translation requests even in the IOMMU case, since the IOTLB can filter part of them.

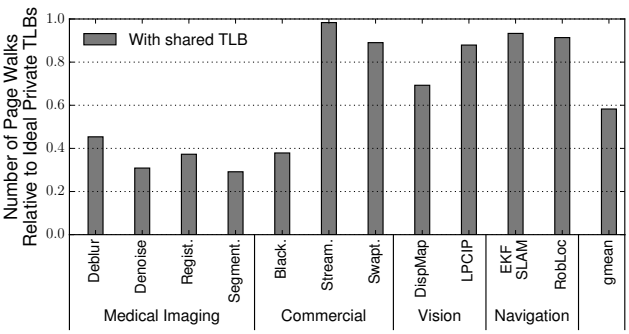

Figure 12: Page walk reduction from adding a 512-entry shared TLB to infinite-sized private TLBs

To further isolate the effect of page sharing caused by data tiling, we run simulations with infinite-sized private TLBs so that the capacity issue is eliminated. Figure 12 shows the page walk reduction by adding a 512-entry shared TLB to infinite-sized private TLBs. As infinite-sized private TLBs leave only cold misses, the shared TLB exploits page sharing among those misses and filters duplicate ones, resulting in a 41.7% reduction on average. Notice that not all benchmarks greatly benefit from having a shared TLB, which is due to the different tiling mechanism of each application. While developing a TLB-aware tiling mechanism could potentially reduce duplicate TLB misses, it is not easy to do so when the input data size and tile size are user-defined and thus can be arbitrary. We leave this for future work.

The reminder of the page walks is due to cold TLB misses, where alternating TLB sizes or organization can not make a difference. Therefore, we propose an efficient page walk handling mechanism to minimize the latency penalty introduced by those page walks.

4.4 Host Page Walks

As the IOMMU is not capable of delivering efficient page walks, the performance of accelerators still suffer from excessive long page walk latency even with reduced number of page walks. While providing a dedicated full-blown MMU support for accelerators could potentially alleviate this problem, there may not be a need to establish a new piece of hardware especially when off-the-shelf resources can be readily leveraged by accelerators.

This opportunity lies in the coordination between the accelerators and the host core that launches them. After the computation task has been offloaded from the host core to accelerators, a common practice is to put the host core into spinning so that the core can react immediately to any status change of the accelerators. As a result, during the execution of accelerators, the host core MMU and data cache is less stressed, which can be utilized to service translation requests from the accelerators invoked by this core. By offloading page walk operations to the host core MMU, the following benefits can be achieved:

First, the MMU cache support is provided from the host core MMU. Commercial CPUs have introduced MMU caches to store upper level entries in page walks [6, 27]. The page walker accesses the MMU cache to determine if one or more levels of walks can be skipped before issuing memory references. As characterized in Section 2.2, accelerators have extremely regular page access behaviors with small page divergence. Therefore, the MMU cache can potentially work very well with accelerators by capturing the good locality in upper levels of the page table. We expect that the MMU cache is able to skip all three non-leaf page table accesses for the majority of time, leaving only one memory reference required for each page walk. We assume an AMD-style page walk cache [6] in this paper, which stores entries in a data cache fashion. However, other implementations such as Intel's paging structure cache [27] could provide similar benefits.

Second, PTE (page table entry) locality within a cacheline provides an opportunity to amortize the cost of memory accesses over more table walks. Unlike the IOMMU, the CPU MMU has data cache support, which means a PTE is first brought from the DRAM to the data cache and then to the MMU. Future access to the same entry, if misses in the MMU cache, could still hit in the data cache with much lower latency than a DRAM access. More importantly, as one cacheline could contain eight PTEs, one DRAM access for a PTE potentially prefetches seven consecutive ones, so that future references to these PTEs could be cache hits. While this may not benefit CPU or GPU applications with large page divergence, we have shown that the regularity of accelerator TLB misses could permit improvement through prefetching.

Third, resources are likely warmed up by previous host core operations within the same application's virtual address space. Since a unified virtual address space permits a close coordination between the host core and the accelerators, both can work on the same data with either general-purpose manipulation or high-performance specialization. Therefore, the host core operations could very well warmup the resources for accelerators. Specifically, a TLB miss triggered by the host core brings both upper-level entries to the MMU cache and PTEs to the data cache, leading to reduced page walk latency for accelerators in the near future. While the previous two benefits can also be obtained through any dedicated MMU with an MMU cache and data cache, this benefit is unique to host page walks.

4.4.1 Implementation

Modifications to accelerator TLBs. In addition to the accelerator ID bits in each TLB entry, the shared TLB also needs to store the host process (or context) ID within each entry. On a shared TLB miss, a translation service request with the virtual address and the process ID is sent through the interconnect to the host core operating within the same virtual address space.

Modifications to host MMUs. The host core MMU must be able to distinguish accelerator page walk requests from the core requests, so that PTEs can be sent back to the accelerator shared TLB instead of being inserted into the host core TLBs after page walking. As CPU MMUs are typically designed to handle one single page walk at a time, a separate port and request queue for accelerator page walk requests are required

to buffer multiple requests. An analysis on the number of outstanding shared TLB misses is presented in Section 5.1. Demultiplexer logic is also required for the MMU to send responses with the requested PTE back to the accelerator shared TLB.

Correctness issues. In contrast to implementing a dedicated MMU for accelerators where coordination with the host core MMU is required on page fault handling, our approach requires no additional support for system-level correctness issue. If a page walk returns a NULL pointer on the virtual address requested by accelerators, the faulting address is written to the core's CR2 register and an interrupt is raised. The core can proceed with normal page fault handling process without the knowledge of the requester of the faulting address. The MMU is signaled once the OS has written the page table with the correct translation. And then the MMU finishes the page walk to send the requested PTE to the accelerator shared TLB. The support for TLB shootdowns work the same as in the IOMMU case.

4.4.2 Evaluation

To evaluate the effects of host page walks, we simulate an 8KB page walk cache with 3-cycle access latency, and a 2MB data cache with 20-cycle latency. If the PTE request misses in the data cache, it is forwarded to the off-chip DRAM which typically takes more than 200 cycles. We faithfully simulate the interconnect delays in a mesh topology.

We first evaluate the capability of the host core MMU by showing the average latency of page walks that are triggered by accelerators. Figure 13 shows that the host core MMU consistently provides low-latency page walks across all benchmarks, with an average of only 58 cycles. Given the latency of four consecutive data cache accesses is 80 cycles plus interconnect delays, most page walks should be a combination of MMU cache hits and data cache hits, with DRAM access only in rare cases. This is partly due to the warmup effects where cold misses in both MMU cache and data cache are minimized. Based on this, it is difficult for a dedicated MMU to provide even lower page walk latency than the host core MMU.

We further analyze the average translation latency of each design to relate to our latency sensitivity study. As shown in Figure 14(a), the average translation latency across all benchmarks for designs with private TLBs and two-level TLB is 101.1 and 27.7 cycles, respectively. This level of translation latency, if uniformly distributed, should not result in more than 50% performance slowdown according to Figure 7. However, as shown in Figure 14(b), the average translation latency of the requests that trigger page walks is well above 1000 cycles for the two designs that uses an IOMMU. This is due to both long page walk latency and queueing latency when there are multiple outstanding page walk requests. With such long latencies added to the runtime, accelerators become completely ineffective in latency hiding, even on shorter latencies which could otherwise be tolerated by double buffering. In contrast, host page walks reduce page walk latencies and mean-

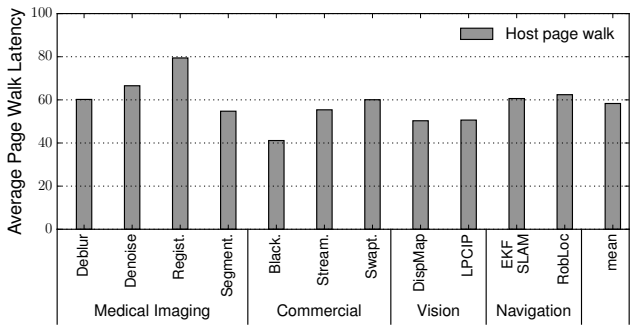

Figure 13: Average page walk latency when offloading page walks to the host core MMU

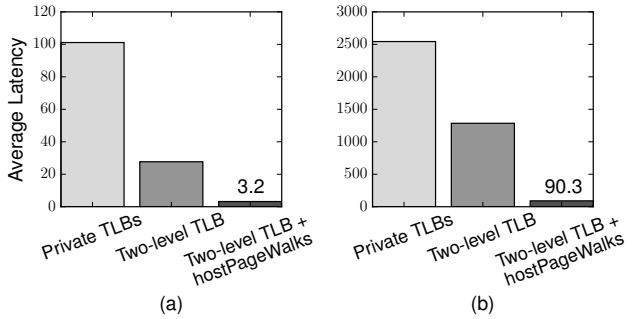

Figure 14: Average translation latency of (a) all requests; (b) requests that actually trigger page walks

while minimize the variance of address translation latency. Therefore, the overall performance benefits from a much lower average address translation latency (3.2 cycles) and decreased level of variations.

4.5 Summary: Two-level TLB and Host Page Walks

Table 3: Configuration of our proposed address translation support

Component	Parameters
Private TLBs	32-entry, 1-cycle access latency
Shared TLB	512-entry, 3-cycle access latency
Host MMU	4KB page, 8KB page walk cache [8]
Interconnect	Mesh, 4-stage routers

Overall design. In summary, to provide an efficient address translation support for accelerator-centric architectures, we first enhance the IOMMU approach by designing a low-latency private TLB for each accelerator. Second, we present a shared level-two TLB design to enable page sharing between accelerators, reducing duplicate TLB misses. The two-level TLB design effectively reduces number of page walks by 76.8%. Finally, we propose to offload page walk request to the host core MMU so that we can efficiently handle page walks with an average latency of 58 cycles. Table 3 summarizes the parameters of key components in our design.

Overall system performance. Figure 15 compares the performance of different designs against the ideal address translation. Note that the first three designs rely on the IOMMU for page walks which could take more than 900 cycles. Our proposed three designs, as

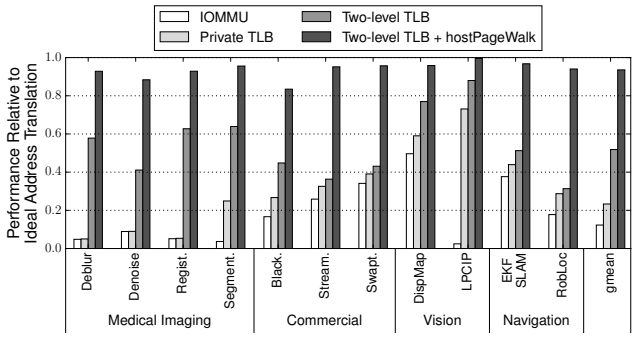

Figure 15: Total execution time normalized to ideal address translation

shown in Figure 15 achieve 23.3%, 51.8% and 93.6% of the ideal address translation performance, respectively, while the IOMMU baseline only achieves 12.3% of the ideal performance. The performance gap between our combined approach (two-level TLB with host page walks) and the ideal address translation is reduced to 6.4% on average, which is in the range deemed acceptable in the CPU world (5-15% overhead of runtime [6, 7, 10, 41]).

5. DISCUSSION

5.1 Impact of More Accelerators

While we have shown that significant performance improvement can be achieved for four accelerator instances by sharing resources including the level-two TLB and host MMU, it is possible that resource contention with too many sharers results in performance slowdown. Specifically, since CPU MMUs typically handle one page walk at a time, the host core MMU can potentially become a bottleneck as the number of outstanding shared TLB misses increases. To evaluate the impact of launching more accelerators by the same host core, we run simulations with 16 accelerator instances in the system with the same configuration summarized in Table 3.

We compare the average number of outstanding shared TLB misses[4] for the 4-instance and 16-instance cases in Figure 16. Our shared TLB provides consistent filtering effect, requiring on average only 1.3 and 4.9 outstanding page walks at the same time in the 4-instance and 16-instance cases, respectively. While more outstanding requests lead to longer waiting time, subsequent requests are likely to hit in the page walk cache and data cache due the regular page access pattern, thus requiring less service time. Using a dedicated MMU with threaded page walker [45] could reduce the waiting time. However, the performance improvement may not justify the additional hardware complexity even for GPUs [44].

Figure 17 presents the overall performance of our proposed address translation support relative to the ideal

[4]As TLB misses are generally sparse during the execution, we only sample the number when there is at least one TLB miss.

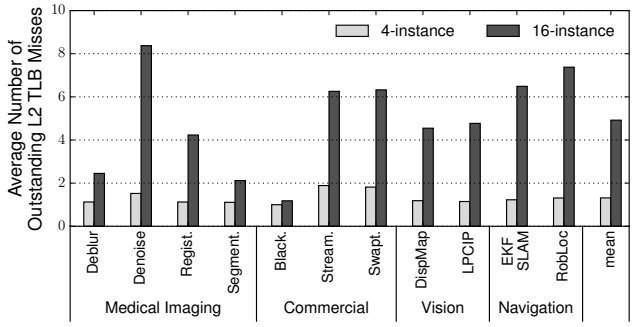

Figure 16: The average number of outstanding shared TLB misses of the 4-instance and 16-instance cases

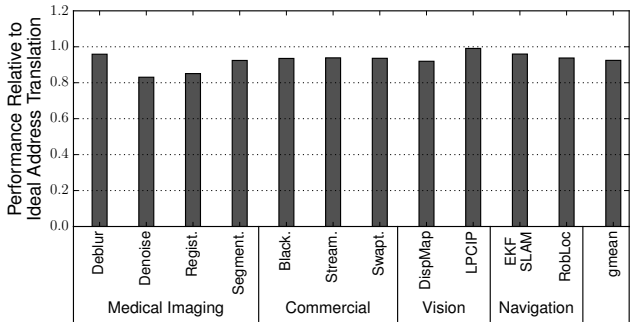

Figure 17: Performance of launching 16 accelerator instances relative to ideal address translation

address translation when there are 16 accelerator instances. We can see that even with the same amount of shared resource, launching 16 accelerator instances does not have a significant impact over the efficiency of address translation, with the overhead being 7.7% on average. While even more active accelerators promise greater parallelism, we already observe diminishing returns in the 16-instance case, as the interconnect and memory bandwidth is saturating.

Another way of having more active accelerators in the system is by launching multiple accelerators using multiple CPU cores. However, the page walker in each core MMU will not experience higher pressure in such scenario since our mechanism requires that accelerators only offload TLB misses to the host core that operates within the same application's virtual address space. A larger shared TLB may be required for more sharers where a banked placement could be more efficient. We leave this for future work.

5.2 Large Pages

Large pages [53] can potentially reduce TLB misses by enlarging TLB reach and speedup misses by requiring less accesses to memory while walking the page table. To reduce memory management overhead, the OS with Transparent Huge Page [4] support can automatically construct large pages by allocating contiguous baseline pages aligned at the large page size. As a result, developers no longer need to identify the data that could benefit from using large pages and explicitly request the allocation of large pages.

As we have shown that accelerators typically feature

bulk transfers of consecutive data and are sensitive to long memory latencies, large pages are expected to improve the overall performance of accelerators by reducing TLB misses and page walk latencies. We believe this approach is orthogonal to ours and can be readily applied to the proposed two-level TLB and host page walk design. It is worthwhile to note that the page sharing effect resulted from tiling of high-dimensional data will become more significant under large pages, leading to increased number of TLB misses on common pages. Our shared TLB design is shown to be effective in alleviating this issue.

6. RELATED WORK

Address Translation on CPUs. To meet the ever increasing memory demands of memory intensive applications, commercial CPUs have included one or more levels of TLBs [23, 50] and private low-latency caches [8, 27] in the per-core MMU to accelerate address translation. These MMU caches have been shown to greatly increase performance for CPU applications across different implementations [6]. TLB augmentation has long been studied in the community. Prefetching [29, 32, 48] techniques are proposed to speculate on PTEs that will be referenced in the future. While such techniques benefit applications with regular page access patterns, additional hardware such as a prefetching table is typically required. Shared last-level TLBs [10] and shared MMU caches [9] are proposed for multicores to accelerate multithreaded applications by sharing translations between cores. The energy overheads of TLB resources is also studied [33], advocating for energy-efficient TLBs.

Address Translation on Fused CPU-GPUs. A recent IOMMU tutorial [34] presents a detailed introduction to the IOMMU design within the AMD fused CPU-GPUs, with a key focus on its functionality and security. Though it also enables translation caching in devices, no detail or quantitative evaluation is revealed. To improve address translation on fused CPU-GPUs, [44, 45] propose GPU MMU designs consisting of post-coalescer TLBs and logic to walk the page table. As GPUs can potentially require hundreds of translations per cycle due to high parallelism in the architecture, [44] uses 4-ported private TLBs and improved page walk scheduling whereas [45] uses highly threaded page walker to serve bursts of TLB misses.

Based on our characterization of customized accelerators, we differentiate the address translation requirements between customized accelerators and GPUs in three ways. First, accelerators do not use instructions and have much more regular consecutive access patterns compared to GPUs, which enables a simpler private TLB design. Second, the page sharing effect between accelerators cannot be resolved using the same coalescing structure as in GPU since accelerators are not designed to execute in lockstep. Instead, a shared TLB design is tailored to compensate the impact of data tiling. Third, while GPUs average 60 concurrent TLB misses [45], we have shown that accelerators have far less outstanding TLB misses, below 5 on average, even with 16 active accelerator instances in the system, after filtering by the two-level TLB. Therefore, host page walks with the existing MMU cache and data cache support suffice to provide low page walk latency, with an average of only 58 cycles.

Current Virtual Address Support for Accelerators. In order to reap the benefits of unified address space, some initial efforts have been made to support address translation for customized accelerators. Intel-Altera heterogeneous architecture research platform [26] introduces a static 1024-entry TLB with 2MB page size to support virtual address for user-defined accelerators. Similar approach is also adopted in the design of a Memcached accelerator [37]. Such static TLB approach requires allocation of pinned memory and kernel driver intervention on TLB refills. As a result, programmers need to work with special APIs and manually manage various buffers, which can be a giant pain.

Xilinx Zynq SoC [1] provides a coherent interface between the ARM Cortex-A9 processor and FPGA programmable logic through the accelerator coherency port. While prototypes in [15, 42] are based on this platform, the address translation mechanism is not detailed. [38] assumes a system MMU support for the designed hardware accelerator. However, the impact on performance is not studied.

Intel and AMD have equipped their commercial processors with IOMMUs [2, 28, 34] to provide address translation support for loosely-coupled devices including customized accelerators and GPUs. rIOMMU [40] improves the throughput for devices that employ circular ring buffers such as network and PCIe SSD controllers, but is not intended for customized accelerators with complex memory behaviors. While we choose an IOMMU configuration as the baseline in this paper for its generality, the key insights of this work are applicable to other platforms with modest adjustments.

7. CONCLUSION

The goal of this paper is to provide simple but efficient address translation support for accelerator-centric architectures. We propose a two-level TLB design and host page walks tailored to the specific challenges and opportunities of customized accelerators. We find that a relatively small and low-latency private TLB with 32 entries for each accelerator reduces page walks by 30.4% compared to the IOMMU baseline. Adding a shared 512-entry TLB eliminates 75.8% in total of page walks by exploiting page sharing resulted from data tiling. Moreover, by simply offloading page walk requests to the host core MMU, the average page walk latency can be reduced to 58 cycles. Our evaluation shows that the combined approach achieves 93.6% of the performance of the ideal address translation.

This paper is the first to provide hardware support for a unified virtual address space between the host CPU and customized accelerators with marginal overhead. We hope that this paper could stimulate future research in this area and facilitate the adoption of customized accelerators.

8. REFERENCES

[1] "Zynq-7000 all programmable soc," http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

[2] *AMD I/O Virtualization Technology (IOMMU) Specification*, Advanced Micro Devices, Inc., 2011.

[3] N. Amit, M. Ben-Yehuda, and B.-A. Yassour, "Iommu: Strategies for mitigating the iotlb bottleneck," in *Workshop on Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2010.

[4] A. Arcangeli, "Transparent hugepage support," in *KVM Forum*, 2010.

[5] *ARM System Memory Management Unit Architecture Specification*, ARM Ltd., 2015.

[6] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: Skip, don't walk (the page table)," in *ISCA-37*, 2010.

[7] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *ISCA-40*, 2013.

[8] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *ASPLOS-XIII*, 2008.

[9] A. Bhattacharjee, "Large-reach memory management unit caches," in *MICRO-46*, 2013.

[10] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level tlbs for chip multiprocessors," in *HPCA-17*, 2011.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *PACT-17*, 2008.

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[13] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill, "Translation lookaside buffer consistency: A software approach," in *ASPLOS-III*, 1989.

[14] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern cpu-fpga platforms," in *DAC-53*, 2016.

[15] E. S. Chung, J. D. Davis, and J. Lee, "Linqits: Big data on little clients," in *ISCA-40*, 2013.

[16] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[17] J. Cong, Z. Fang, M. Gill, and G. Reinman, "Parade: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration," in *ICCAD*, 2015.

[18] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman, "Accelerator-rich architectures: Opportunities and progresses," in *DAC-51*, 2014.

[19] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich cmps," in *DAC-49*, 2012.

[20] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for domain-specific accelerator-rich cmps," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, pp. 131:1–131:26, Apr. 2014.

[21] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ISCA-38*, 2011.

[22] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, Sep. 2012.

[23] P. Hammarlund, "4th generation intel core processor, codenamed haswell," in *Hot Chips*, 2013.

[24] J. Hestness, S. W. Keckler, and D. A. Wood, "A comparative analysis of microarchitecture effects on cpu and gpu memory system behavior," in *IISWC*, 2014.

[25] *HSA Platform System Architecture Specification 1.0*, HSA Foundation, 2015.

[26] *Accelerator abstraction layer software programmerâĂŹs guide*, Intel Corporation.

[27] *TLBs, Paging-Structure Caches, and Their Invalidation*, Intel Corporation, 2008.

[28] *Intel Virtualization Technology for Directed I/O Architecture*, Intel Corporation, 2014.

[29] B. L. Jacob and T. N. Mudge, "A look at several memory management units, tlb-refill mechanisms, and page table organizations," in *ASPLOS-VIII*, 1998.

[30] C. Johnson, D. Allen, J. Brown, S. VanderWiel, R. Hoover, H. Achilles, C.-Y. Cher, G. May, H. Franke, J. Xenedis, and C. Basso, "A wire-speed powertm processor: 2.3ghz 45nm soi with 16 cores and 64 threads," in *ISSCC*, 2010.

[31] T. Johnson and U. Nawathe, "An 8-core, 64-thread, 64-bit power efficient sparc soc (niagara2)," in *ISPD*, 2007.

[32] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for tlb prefetching: An application-driven study," in *ISCA-29*, 2002.

[33] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Energy-efficient address translation," in *HPCA-22*, 2016.

[34] A. Kegel, P. Blinzer, A. Basu, and M. Chan, "Virtualizing io through io memory management unit (iommu)," in *ASPLOS-XXI Tutorials*, 2016.

[35] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *MICRO-46*, 2013.

[36] A. Krishna, T. Heil, N. Lindberg, F. Toussi, and S. VanderWiel, "Hardware acceleration in the ibm poweren processor: Architecture and performance," in *PACT-21*, 2012.

[37] M. Lavasani, H. Angepat, and D. Chiou, "An fpga-based in-line accelerator for memcached," *Computer Architecture Letters*, vol. 13, no. 2, pp. 57–60, 2014.

[38] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing soc accelerators for memcached," in *ISCA-40*, 2013.

[39] M. Lyons, G.-Y. Wei, and D. Brooks, "Multi-accelerator system development with the shrinkfit acceleration framework," in *ICCD*, 2014.

[40] M. Malka, N. Amit, M. Ben-Yehuda, and D. Tsafrir, "riommu: Efficient iommu for i/o devices that employ ring buffers," in *ASPLOS-XX*, 2015.

[41] C. McCurdy, A. L. Coxa, and J. Vetter, "Investigating the tlb behavior of high-end scientific applications on commodity microprocessors," in *ISPASS*, 2008.

[42] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "Snnap: Approximate computing on programmable socs via neural acceleration," in *HPCA-21*, 2015.

[43] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *MICRO-42*, 2009.

[44] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces," in *ASPLOS-XIX*, 2014.

[45] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of gpu lanes," in *HPCA-20*, 2014.

[46] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency and flexibility in specialized

computing," in *ISCA-40*, 2013.

[47] R. Sampson, M. Yang, S. Wei, C. Chakrabarti, and T. F. Wenisch, "Sonic millip3de: A massively parallel 3d-stacked accelerator for 3d ultrasound," in *HPCA-19*, 2013.

[48] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based tlb preloading," in *ISCA-27*, 2000.

[49] P. Schaumont and I. Verbauwhede, "Domain-specific codesign for embedded security," *IEEE Computer*, vol. 36, no. 4, pp. 68–74, April 2003.

[50] M. Shah, R. Golla, G. Grohoski, P. Jordan, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja, "Sparc t4: A dynamically threaded server-on-a-chip," *IEEE Micro*, vol. 32, no. 2, pp. 8–19, Mar. 2012.

[51] Y. S. Shao, S. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Toward cache-friendly hardware accelerators," in *Proc. Sensors to Cloud Architectures Workshop (SCAW), in conjuction with HPCA-21*, 2015.

[52] P. Stillwell, V. Chadha, O. Tickoo, S. Zhang, R. Illikkal, R. Iyer, and D. Newell, "Hippai: High performance portable accelerator interface for socs," in *HiPC*, 2009.

[53] M. Talluri and M. D. Hill, "Surpassing the tlb performance of superpages with less operating system support," in *ASPLOS-VI*, 1994.

[54] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *ASPLOS-XV*, 2010.