Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems

Jan Vesely[#*], Arkaprava Basu[#], Mark Oskin[#], Gabriel H. Loh[#] Abhishek Bhattacharjee*

[#]AMD Research
Advanced Micro Devices, Inc.
{jan.vesely, arkaprava.basu, mark.oskin, gabriel.loh}@amd.com

*Department of Computer Science
Rutgers University
abhib@cs.rutgers.edu

Abstract – **Computing is becoming increasingly heterogeneous with accelerators like GPUs being tightly integrated with CPUs on the same die. Extending the CPU's virtual addressing mechanism to these accelerators is a key step in making accelerators easily programmable. In this work, we analyze, using real-system measurements, shared virtual memory across the CPU and an integrated GPU. We make several key observations and highlight consequent research opportunities: (1) servicing a TLB miss from the GPU can be an order of magnitude slower than that from the CPU and consequently it is imperative to enable many concurrent TLB misses to hide this larger latency; (2) divergence in memory accesses impacts the GPU's address translation more than the rest of the memory hierarchy, and research in designing address translation mechanisms tolerant to this effect is imperative; and (3) page faults from the GPU are considerably slower than that from the CPU and software-hardware co-design is essential for efficient implementation of page faults from throughput-oriented accelerators like GPUs. We present a detailed measurement study of a commercially available integrated APU that illustrates these effects and motivates future research opportunities.**

I. Introduction

Computing has witnessed a proliferation of accelerators serving a diverse set of needs like cryptography, graphics, databases, regular expressions [2], [12], [14]. Accelerators often enable significantly superior performance and power efficiency for specific tasks, compared to general-purpose CPUs. Consequently, accelerators are increasingly being integrated with the CPU on the same die. Such tight integration allows efficient offloading of computation to accelerators. Commercial manufacturers like AMD, Apple, Intel, and Qualcomm, today ship millions of processors with CPUs and GPUs tightly coupled together on a single die. IBM's CAPI [24] and ARM®'s ACP [11] extend this concept to a plethora of third-party accelerators.

A key challenge in harnessing the full potential of accelerators in these heterogeneous processors is to make them easily programmable. A crucial first step in addressing this challenge is to enable shared (unified) coherent virtual memory across CPUs and accelerators. Indeed, industry-promoted models such as the Heterogeneous System Architecture (HSA) [26] mandate shared virtual memory in an effort to make accelerators a first-class computing element. This allows a pointer on the CPU to be equally valid on an accelerator; thus avoiding manual data copies. Importantly, it helps provide a programming environment familiar to common programmers.

In this work, we present a detailed characterization and analysis of a shared virtual memory system across the CPU and the integrated GPU in a commercially available heterogeneous processor. To the best of our knowledge, this is the first such study on a commercial platform. We study how the CPU's virtual memory is extended to the integrated-GPU in three key aspects: (1) virtual-to-physical address translation; (2) creation of new virtual-to-physical address mappings through page faults; and (3) invalidation of stale address mappings through TLB (Translation Lookaside Buffer) shootdowns. We use six applications that use the integrated-GPU to perform computation. Although we focus on the GPU, our key results are applicable to other throughput-orientated accelerators.

Our measurements highlight several new research opportunities in this space. (1) A TLB miss on the GPU can be 25× slower than that on the CPU. Research into enabling greater concurrency in servicing TLB misses from throughput-oriented accelerators is critical to hide this latency; (2) Poor locality in memory accesses, particularly from divergent accesses within the same wavefront or warp, impacts the GPU's address translation hardware more than the rest of the memory hierarchy. Research into divergence-tolerant address translation mechanisms is important; (3) The prefetching of address translations built into the industry standard PCIe® specification may hurt performance under certain circumstances. More research into effective translation prefetching for accelerators is necessary; (4) Servicing GPU page faults can be significantly slower than that for CPU page faults. Most of this added time is spent in the operating system (OS) kernel, and likely can be designed out; (5) The latency of a TLB shootdown on the GPU is comparable to that in the CPU. Research into reducing TLB shootdown latency in heterogeneous systems needs to optimize both the CPU and GPU shootdown. Several other such observations and research opportunities are detailed throughout this work.

Section II provides background on how integrated-GPUs perform basic virtual-memory operations. Section III describes our methodology and applications used. Section IV describes our analysis of address translation while Section V and Section VI explore page fault and shootdown costs.

II. Background: Shared Virtual Memory

Figure 1 depicts the system that we analyze in this work. A hardware block called the IO Memory Management Unit (IOMMU) services the address translation requests of the

978-1-5090-1953-3/16/$31.00 ©2016 IEEE

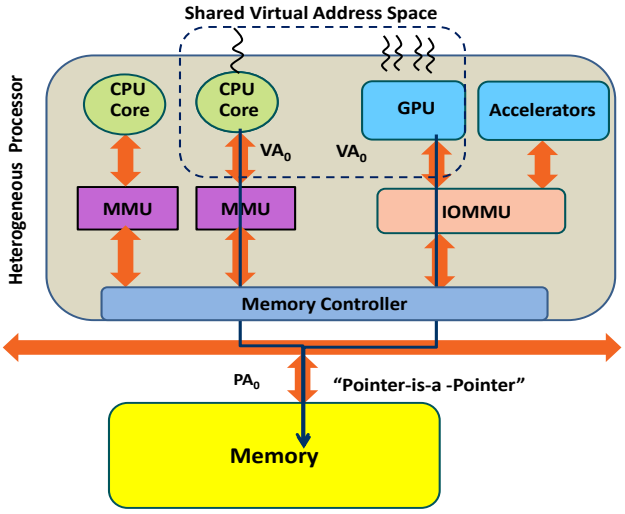

Figure 1. Heterogeneous system enabling shared virtual

GPU and other accelerators. The IOMMU resides in the processor's northbridge complex. The IOMMU can access the same x86-64 page table structures used by processes running on the CPU. This enables the accelerator to share the same set of page tables (and thus the same virtual address space) as the processes running on the CPU via the IOMMU. A software driver in the OS executing on the CPU manages the IOMMU. The runtime software, in coordination with the OS driver, sets up the IOMMU to enable accelerators access to the same virtual address spaces of the CPU. In the following subsections we describe how basic virtual-memory operations are performed by the integrated-GPU in this environment. Although the following operations are explained for the GPU, similar mechanisms are applicable to other accelerators and IO devices.

A. GPU address translation

The GPU has its own TLB hierarchy that caches recently used address translations. On a GPU TLB miss, a translation request is sent as an ATS (Address Translation Service [17]) request packet over the PCIe®-based [27] internal interconnect to the IOMMU. This interconnect carries PCIe® packets but latency and bandwidth are not necessarily constrained by PCIe®'s electrical specifications. The IOMMU has its own TLB hierarchy which is checked first; on a miss there, a hardware page table walker in the IOMMU checks the page table. ATS requests are tagged with a process address space identifier (PASID) and the IOMMU maintains a table that matches PASIDs to page table base physical addresses. Once the address is successfully translated, the IOMMU sends an ATS response to the GPU. The protocol and packet formats for ATS requests and responses are part of the PCIe® standard specification and are the same across all accelerators.

The PCIe®'s ATS protocol enables devices (and accelerators) to prefetch translation requests for up to eight *contiguous* virtual address pages in a single ATS response from the IOMMU. By default, the GPU in our system allows the prefetch value to the maximum setting of eight.

Comparison with CPUs: In the CPU, per-core Memory management Units (MMUs) are responsible for address translations. In contrast, the IOMMU services requests from all accelerators. Unlike the CPU's MMU, the IOMMU is not tightly integrated with CPU's data cache hierarchy. The data caches may contain the most up-to-date translations but the cached copies cannot be directly accessed by accelerators.

B. GPU page faults

If the IOMMU's page table walker fails to find the desired translation in the page table, it sends an ATS response to the GPU notifying it of this failure. This in turn corresponds to a page fault. In response, the GPU sends *another* request to the IOMMU called a Peripheral Page Request (PPR). The IOMMU places this request in a memory-mapped queue and raises an interrupt on the CPU. Multiple PPR requests can be queued before the CPU is interrupted. The OS must have a suitable IOMMU driver to process this interrupt and the queued PPR requests. In Linux, while in an interrupt context, the driver pulls PPR requests from the queue and places them in a work-queue for later processing. Presumably this design decision was made to minimize the time spent executing in an interrupt context, where lower priority interrupts would be disabled. At a later time, an OS worker-thread calls back into the driver to process page fault requests in the work-queue. Once the requests are serviced, the driver notifies the IOMMU. In turn, the IOMMU notifies the GPU. The GPU then sends *another* ATS request to retry the translation for the original faulting address.

Comparison with CPU: On the CPU, a hardware exception is raised on a page fault, which immediately switches to the OS. In most cases in Linux, this routine services the page fault directly, instead of queuing it for later processing. Contrast this with a page fault from an accelerator, where the IOMMU has to interrupt the CPU to request service on its behalf, and also note the several back-and-forth messages between the accelerator, the IOMMU, and the CPU. Furthermore, page faults on the CPU are generally handled one at a time on the CPU, while for the GPU they are batched by the IOMMU and OS work-queue mechanism.

C. GPU TLB shootdowns

The IOMMU plays a pivotal role in extending the TLB shootdown process to GPUs. An OS driver monitors any changes to virtual address mappings for address spaces shared with the GPU and triggers a TLB shootdown when necessary. The driver first sends a command to the IOMMU via a memory-resident command queue to invalidate the stale mapping in the IOMMU's TLB hierarchy. The driver then waits for the IOMMU to confirm successful invalidation from the IOMMU. Next, the driver commands the GPU (via the IOMMU) to invalidate the stale mapping from its TLB. The IOMMU hardware collects the completion notification of the invalidation in the GPU's TLB and forwards this information to the OS. Note there can be two types of invalidation requests: (1) requests to invalidate a given address mapping in the TLBs, and (2) requests to flush all entries for a given address space.

Comparison with CPU: Historically, CPUs have used a variety of mechanisms to perform remote TLB shootdowns. For example, x86 processors use inter-processor interrupts (IPIs) to keep per-core TLBs coherent. CPUs initiating TLB shootdowns send IPIs to other CPUs in the system that may have a stale entry. The IPI invokes the operating system on those CPUs, which executes a handler to invalidate the local per-CPU TLB. Just as the operating system can choose not to send IPIs to processors that provably cannot have a stale mapping (e.g., they never executed the process), an efficiently constructed driver will selectively send shootdowns to the IOMMU and the accelerators only if the address space was shared.

In contrast, processors like ARM® and PowerPC® have also employed dedicated TLB shootdown instructions in their ISAs. In these cases, software on the shootdown initiator core executes a TLB invalidation instruction, which is then broadcast to the other cores. Although the OS is typically not invoked in this approach, the downside is that broadcast signals are often conservatively relayed to all system cores, constraining the scalability of this approach.

III. Methodology and Workloads

We run our experiments on a system with an AMD A10-7850K APU (previously code-named "Kaveri") as described in Table 1. AMD A10-7850K APU is one of the first heterogeneous processor to support shared virtual memory across the CPU and GPU. We measured TLB events and page table walks using hardware performance counters. We designed a software profiler to access these counters. We also instrumented the Linux IOMMU driver to measure the latency for software events like page faults and TLB shootdowns.

Table 1. Description of experimental system.

CPU	AMD A10-7850K APU, maximum core frequency 3.7GHz.
GPU	8 compute units (CUs), maximum core frequency 720 MHz
Memory	DDR-3, 32GB (4×8GB), 1600 MHz
Software	Linux 4.0 with Kernel Fusion Driver (KFD). Heterogeneous System Architecture (HSA) runtime, C++AMP compiler and OpenCL stack on HSA.

We use six applications (described in Table 2) and two targeted micro-benchmarks for this study. All the applications and micro-benchmarks use the on-die integrated-GPU to perform their primary compute. All the data for the applications reside in the system memory (DRAM) and uses shared virtual addressing between the CPU and the GPU.

IV. Address Translation in Shared Virtual Memory

We first analyze intricacies of the GPU address translation under a controlled execution environment with custom micro-benchmarks and then present performance measurements of the applications.

A. *Analyzing GPU's address translation using micro-benchmarks*

We sought to answer two questions in the analysis using a carefully designed micro-benchmark: (1) what is a typical GPU TLB miss latency, and (2) how much concurrency is supported by the hardware in servicing GPU TLB misses? Answers to these questions reveal potential performance bottlenecks, particularly as shared virtual memory is scaled in future heterogeneous systems.

Table 2. Description of applications used.

Application	Description
B+ Tree search (BPT)	Searches 15M keys in a B+tree concurrently on the GPU. The B+tree is pre-generated.
CoMD	Molecular dynamics simulation that evaluates the force acting on an atom due to other atoms in the system. Force potential computation is evaluated on the GPU.
miniAMR	Applies a stencil calculation on the GPU to a dense 3D array.
miniFE	Assembles and solves a sparse linear-system from the steady-state conduction equation [16].
Graph500	BFS traversal on a Kronecker generated graph. Bottom-up traversal uses the GPU.
XSBench	Monte Carlo neutron transport across macroscopic neutron cross sections.

Our micro-benchmark runs a kernel (GPU program) on the integrated-GPU with a varying number of workitems. Each workitem accesses 10,000 different memory locations in a loop and performs a simple computation (XOR) on the data. A stride (parameter) determines the distance (in bytes) between memory locations accessed by two consecutive accesses in each workitem. We execute the micro-benchmark with one workitem with a stride of 64 bytes (cache line size) and again with a stride of 4KB (page size). We use the hardware performance counters to count the number of TLB and cache misses. Table 3 lists the measurements. When executed with a stride of 64 bytes (second row) each memory access incurs a cache miss while every 64th access (4096/64) incurs a TLB miss. With a stride of 4KB, every access incurs a cache miss and a TLB miss (third row). Thus the difference between these two executions is the number of TLB misses. Therefore, we attribute the difference in the runtime between the two runs to the additional TLB misses. We calculate that the latency of servicing a GPU TLB miss that incurs a page walk by the IOMMU to be 582 nanoseconds (last row). We further ran the same micro-benchmark on the CPU with a single thread for comparative analysis. Measurement on the CPU is presented in the last column and we similarly calculate that latency of the CPU's TLB miss to be 23 nanoseconds. Thus, a TLB miss from the GPU is about 25× slower than on a CPU. Several reasons contribute to the longer latency of GPU TLB misses: (1) TLB miss request and responses travel as PCIe® packets to

Table 3. Measurements of TLB miss latency.

Stride (in bytes)	GPU		CPU	
	Number of TLB misses	Running time (in µsec)	Number of TLB misses	Running time (in µsec)
64 (cache line size)	166	12,707	1,412	134
4096 (page size)	10,010	18,444	15,938	477
TLB miss latency (calculated)	(18,444-12,707)/ (10,010 -166) = **0.58 µs**		(477-134)/ (15,938 -1412) = **0.023 µs**	

the IOMMU; (2) more levels (up to four here) of TLBs to check; (3) the IOMMU's page table walker does not have fast access to CPU caches that might have the latest page table entries; and (4) the resulting wavefront must be rescheduled for execution on the GPU.

We then executed the micro-benchmark on the GPU with an increasing number of workitems from 1 to 64 to estimate the concurrency available in servicing TLB misses. Each workitem accesses a distinct set of memory locations and thus offers no opportunity to coalesce the accesses. The maximum number of outstanding TLB misses (and the corresponding page walk requests) at any given time is thus bounded by the number of workitems. Figure 2 shows how the latency experienced by a wavefront on a GPU TLB miss scales with an increasing number of workitems. Note that there is a significant jump in the page walk latency beyond 16 workitems (and thus, 16 outstanding page table walks). This suggests that in our test hardware, the IOMMU allows up to 16 concurrent page table walks, beyond which the latency of page walks increases sharply due to queuing.

B. Measuring GPU's address translation overhead

Next, we run six applications (Table 2) to measure and analyze the overheads of the GPU's address translation on real workloads. We run each application with increasing memory footprints to understand the scalability of the GPU's address translation overheads. We use hardware performance counters to measure the number of GPU TLB misses and the number of accesses to the page table by the IOMMU. We perform the measurements of each application using 4KB (default) and 2MB pages. Larger pages reduce the number of TLB misses and enable us to better understand the performance benefits of reducing TLB misses.

Figure 3 depicts a summary of our measurements for each application. The x-axis is the approximate memory footprint. The plot's right y-axis represents the GPU runtime, and the left y-axis is the number of GPU TLB misses and the number of accesses to the page table by the IOMMU per kilo wavefront instructions (PKWI) executed on the GPU. A wavefront instruction is a single-instruction-multiple-data (SIMD) instruction executed by workitems (a.k.a GPU threads) in a given wavefront (warp) in a lock-step fashion. In one set of runs, the applications make use of 4KB pages (4k), and in another set of runs the applications make use of larger 2MB pages (2m).

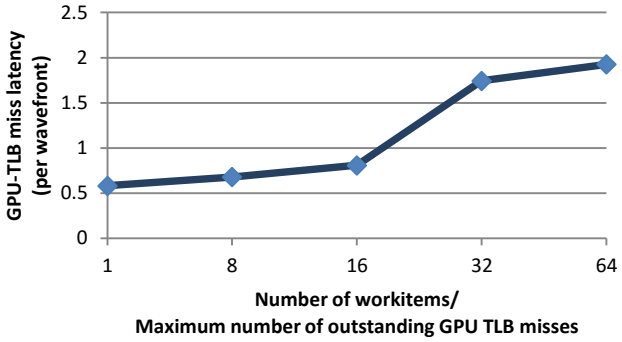

Figure 2. Scaling of GPU TLB miss latency.

For example, we scale the memory footprint of the workload BPT using different input sets, from 2 to 21 GB. Observe that with 4KB pages, the GPU TLB miss rate goes up from 0.5 to 4.4 misses per KWI [gTLB_MPKWI (4k)]. The number of accesses to the page table goes up from 0.4 to 2 accesses per KWI [PTa_PKWI(4k)]. The GPU TLB misses and the number of accesses to the IOMMU becomes negligible if larger 2MB pages are used. Correspondingly, there is up to an 11% reduction in execution time. We note that PTa_PKWI counts the number of accesses to the in-memory page table and not the number of page table walks. In x86-64, a page table walk can incur upto four accesses to the page table.

On the opposite end of the spectrum is graph500 (Figure 3(e)). There are at most 2.5 GPU TLB misses per KWI. A larger page size (2MB) eliminates almost all TLB misses for graph500, but there is no observable change in execution time. This suggests that the GPU's address translation overhead is not a factor in graph500's performance. We find that graph500 loads data from the memory to the GPU's scratchpad or local data store (LDS) in contiguous chunks, and thus amortizes TLB misses well. Similar behavior is observed for miniFE which also makes use of LDS. In contrast, BPT accesses data more randomly and has less opportunity to amortize this cost. Other applications (Figure 3(b)–(e)) show varying degrees of sensitivity to the GPU's address translation overheads.

In summary, BPT, CoMD, and miniAMR are sensisitive to TLB miss rates. BPT, CoMD, and miniAMR, respectively, achieve up to 11.9%, 9.7% to 4.6% improvement in runtimes when TLB misses are significantly reduced. On the other hand, graph500, miniFE, and XSBench are insensitive to translation overheads.

Furthermore, preliminary experiments with a recently released second generation APU suggests that while the overall runtimes of applications have significantly improved, the contribution of overheads due to address translation have doubled. This suggests that with future improvement in the rest of the coherent memory hierarchy the address translation is likely to become the bottleneck, unless paid attention to.

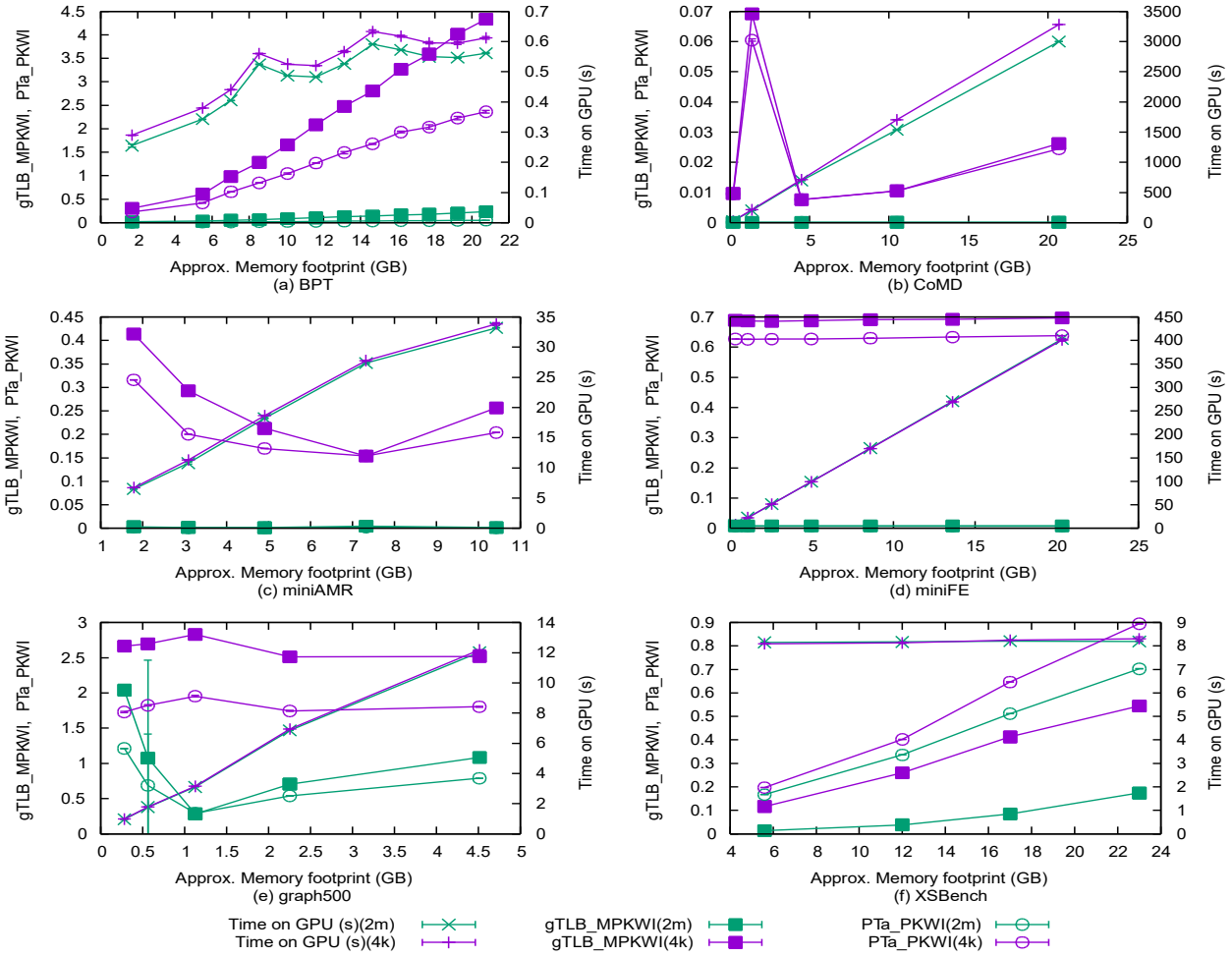

Figure 3. GPU TLB miss rates and impact of larger page size.

C. *Effect of locality on GPU's address translation*

The effect of memory-access locality on shared-memory heterogeneous applications is subtle. With sufficient application concurrency and locality, a small number of TLB misses do not affect performance. On a TLB miss, the GPU can switch to another wavefront (warp) to hide the latency of the resulting page walk. On the other hand, if every wavefront incurs a TLB miss (poor locality), or worse, if many workitems (GPU thread) within a wavefront incur a TLB miss (called "data divergence" in GPU terminology), then programs can be highly sensitive to address-translation overhead.

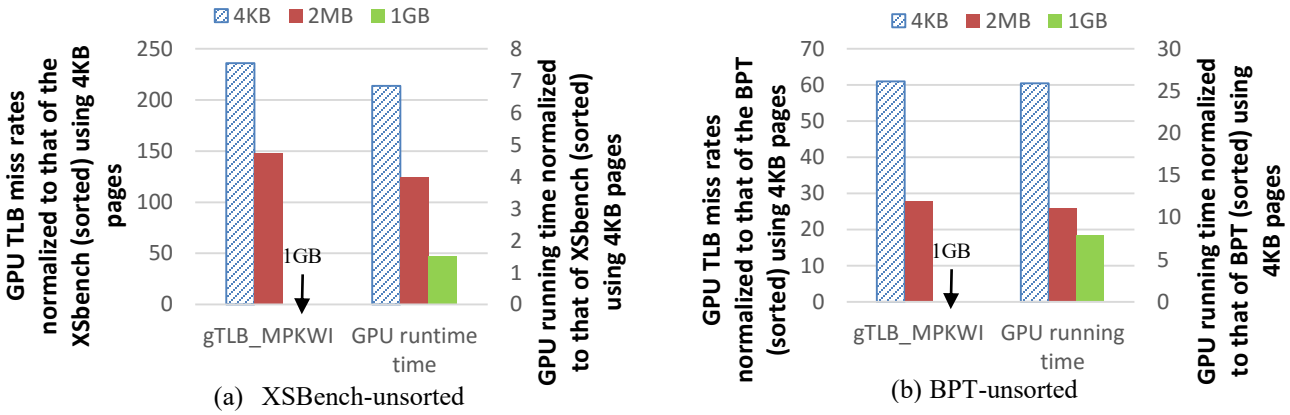

Figure 4. GPU TLB miss rates and running time of unsorted versions of XSBench and BPT.

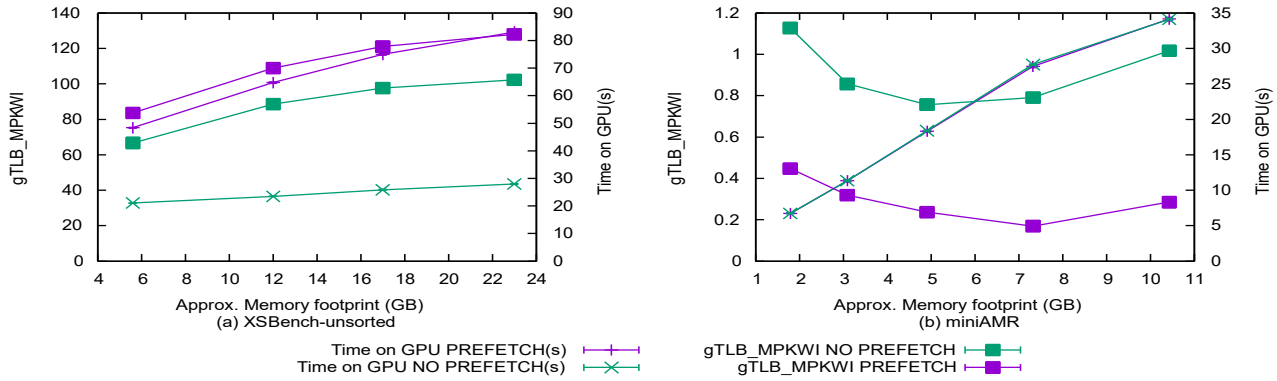

Figure 5. Effect of translation prefetching on GPU's address translation mechanism.

To analyze how poor locality in memory accesses may affect the GPU's address translation, we used alternative versions of XSBench and BPT. Both XSBench and BPT perform concurrent searches on the GPU over a large data structure (e.g., nuclear energy grid and B+ tree, respectively). In the original versions, the search keys are sorted (on the CPU) before performing the searches on the GPU. This significantly increases the locality in the resulting execution because adjacent workitems (GPU threads) in the same wavefront, and wavefronts scheduled nearby in time access the same memory pages. The alternative versions used in this experiment (called XSBench-unsorted and BPT-unsorted) perform the same work but without the pre-sorting step. Thus, XSBench-unsorted and BPT-unsorted demonstrate considerably less locality in their memory accesses.

Figure 4 shows the runtime and the TLB miss rates of XSBench-unsorted and BPT-unsorted normalized to their corresponding original (sorted) versions. In Figure 4(a), the right cluster of bars (using the right y-axis) shows the GPU running time of XSBench-unsorted using different page sizes (4KB, 2MB, 1GB) normalized to the runtime of the original XSBench (sorted) version using only 4KB pages. The left cluster of bars (using the left y-axis) shows the corresponding number for the GPU TLB misses per KWI. Figure 4(b) shows the same data for BPT. Both workloads use the largest dataset for this experiment.

We observe that there is a significant slowdown (6-25×) due to poor locality. This slowdown is due to poor locality in address translation, caching, and DRAM. To isolate the impact on the address translation we use large and huge pages to alleviate TLB misses. For example, observe that with 1GB pages, TLB misses are nearly non-existent (invisible in the graphs) even for the unsorted versions, and correspondingly the slowdown reduces from 6× to 1.5×, and from 26× to 7× for XSBench-unsorted and BPT-unsorted, respectively. This suggests the residual slowdowns (1.5× and 7×) are due to the effect of poor locality in the rest of the memory hierarchy (e.g., caches, memory bandwidth). This strongly indicates that poor locality affects the GPU's address translation far more than the rest of the memory hierarchy. Research into divergence-tolerant address translation mechanisms for throughput-oriented accelerators is important.

D. Effect of address translation prefetching

As described in Section II.A, the GPU by default prefetches translations for up to eight contiguous (4KB) pages in each ATS request sent to the IOMMU. Figure 4 depicts the runtime and the number of GPU TLB misses per KWI with and without translation prefetching for two of the workloads: XSBench-unsorted and miniAMR. In Figure 5(a), we find that XSBench-unsorted incurs up to 24% more GPU TLB misses, and consequently, nearly 3× performance degradation due to translation prefetching. The contiguous prefetches are useless for XSBench-unsorted's nearly random accesses to large amounts of data. Furthermore, the useless prefetches evict useful translations from the GPU TLB and ultimately hurt performance by increasing the number of page table walks. Conversely, we find that translation prefetching reduces the number of TLB misses for the majority of workloads studied. For example, Figure 5(b), presents measurements for miniAMR. We observe that prefetching translations aids miniAMR by reducing the number of TLB misses by half. These measurements suggest that the effectiveness of translation prefetching is highly application-dependent and providing application-aware or programmable prefetching would be prudent.

Observations and opportunities:

1. Latency of servicing a TLB miss is significantly higher on a GPU than on a CPU (~25×).
2. Increasing the number of concurrent page table walks supported by the hardware is key to supporting diverse heterogeneous applications.
3. Half of the programs we studied suffer performance degradation from GPU address translation overheads.
4. Larger pages are effective in reducing TLB misses. Heterogeneous software and hardware should enhance support for larger page sizes.
5. Divergence in memory accesses impacts address translation overhead more than cache and DRAM latency. Research into divergence-tolerant address translation mechanisms for throughput-oriented accelerators is important.
6. Prefetching address translations can degrade performance for programs with poor locality. Application-dependent translation prefetching is desirable.

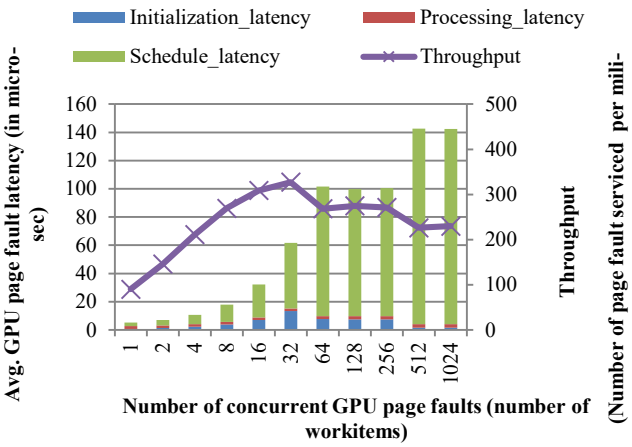

Figure 6. Scaling of GPU page faults.

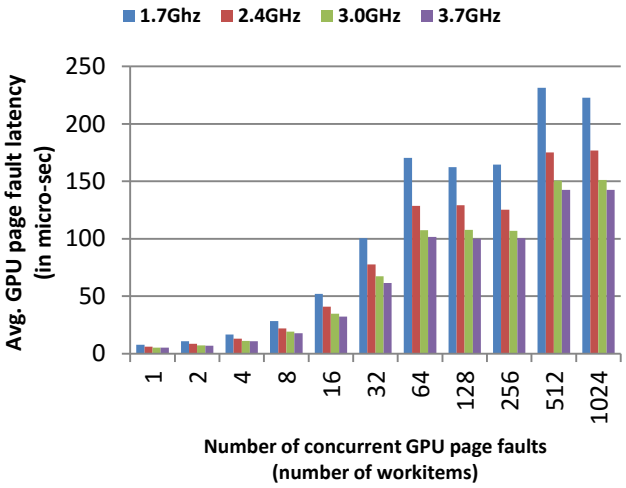

Figure 7. Scaling of GPU page fault with CPU frequency.

V. Extending Page Faults to GPUs

A key feature of virtual memory is the ability to establish mappings between virtual and physical addresses on demand. This defers committing physical memory until (and if) needed. Page faults allow the operating system to consolidate physical memory across multiple concurrently running processes and enable memory over-commitment. This is particularly useful, for example, when an application maps a large input dataset to memory, but only ends up using a small portion of it. Further, page faulting is key in enforcing page permission changes and many advanced memory management techniques like garbage collection and page frame reclamation.

While the ability to perform page faults has been an integral part of a CPU's virtual memory for decades, accelerators like GPUs traditionally lacked such capability. AMD's A10 APUs released in 2014 is one of the first commercial heterogeneous processors to support page faulting on the shared memory from accelerators. However, today's heterogeneous applications are not written to utilize this new capability and instead follow an OpenCL-like programming model. In time this will change, but for now, we modified applications to utilize this functionality to study its impact. We focus on soft-page faults, which do not incur accesses to secondary storage.

A. Analyzing GPU page fault latency and throughput

We analyze the latency and throughput of GPU page faults using a micro-benchmark. We instrumented the IOMMU driver to perform the measurements. The micro-benchmark generates a constant number (512,000) of soft-page faults from the GPU. The soft page faults do not access the storage and are generated by the first access to a page in memory. The micro-benchmark is designed to generate faults in controlled bursts by varying the number of workitems. When the micro-benchmark is executed with 'n' workitems, it generates 'n' concurrent page fault requests from the GPU. Thus increasing the value of 'n' generates larger bursts of concurrent page faults. Figure 6 shows the measured latency and throughput of servicing these GPU page faults. The total height of each bar represents the average latency to service a page fault. The left y-axis represents the page fault latency in microseconds. The right y-axis represents throughput of servicing GPU page faults. For example, with 64 workitems, the average latency to service a page fault is around 100 microseconds and 260 page faults are serviced per millisecond (throughput).

We make two key observations: (1) the average latency to service a page fault increases from 5 microseconds to 140 microseconds with increasing number of concurrent page faults from the GPU; and (2) the throughput of servicing GPU page faults does not scale beyond 32 concurrent page faults. To put these numbers in perspective, we executed the experiment on the CPU (single-threaded) and found the typical page fault latency on the CPU is around 1.7 microseconds. GPU page-faults are 3-80× slower. Larger concurrency in servicing page faults from the GPU can help amortize this high latency.

We breakdown the (software) latency to service a GPU page fault in Figure 6. We divide the time to handle a GPU page fault into three major parts: (1) "initialization", the latency for the OS driver to read the fault requests from the PPR queue and pre-process it; (2) "processing", the latency to find a physical page and update the page table; (3) "schedule", the time between initialization and processing of a page fault request. We observe that only a small fraction of the time is spent in actually processing the work to service a page fault. The OS's scheduling delay introduced by the asynchronous handling of GPU page faults is the primary contributor to the latency. This suggests that page faults from the GPU can be handled more efficiently by modifying the OS driver to handle the faults synchronously whenever possible.

Figure 7 shows the relationship between the CPU's core frequency and the latency to service a GPU page fault. The height of each bar in the clusters represents average GPU-page fault latency with the CPU running at the given frequency. We observe that the latency to service a GPU page fault nearly doubles when the CPU core frequency is reduced from 3.7GHz to 1.7GHz. In general, the graph shows that the GPU page fault latency inversely scales linearly with the CPU's core frequency. This suggests that the CPU's core frequency

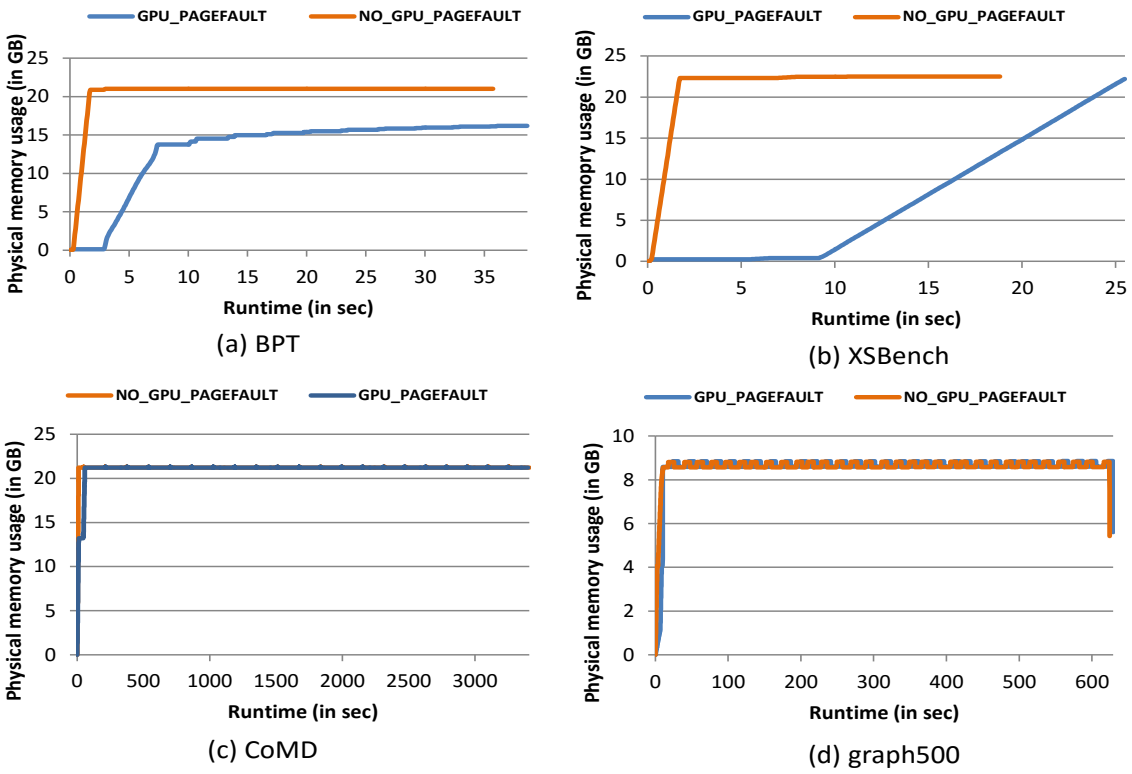

Figure 8. Physical memory usage with and without page faults from GPU.

needs to scale up for faster servicing of GPU page faults and the CPU power setting may affect GPU page fault behavior.

B. Physical memory consolidation through GPU page fault

We measured the reduction in the physical memory footprint through the use of on-demand page faults from the GPU and measure its performance overhead.

We modified four applications (BPT, XSBench, CoMD and Graph500) to utilize demand faulting of memory from the GPU. These modifications include using memory-mapped files, changes to the data structures, and memory allocation. We did not modify miniFE and miniAMR as it was not practical to dynamically fault in data from the GPU to achieve memory consolidation without major alterations to their code bases.

Figure 8 depicts the reduction in physical memory footprint through the use of page faults from the GPU. The x-axis of each graph represents a given workload's running time on the GPU while the y-axis represents the physical memory allocated to the workload at a given time. Each graph has two lines representing two versions of each workload. "GPU_PAGEFAULT", is the modified version of a workload that dynamically faults memory from the GPU. "NO_GPU_PAGEFAULT", represents the original version. The difference between these two lines signifies opportunity to save physical memory. In Figure 8(a), we observe that the physical memory footprint of BPT reduces significantly with GPU page faults. BPT performs concurrent searches on the GPU over a pre-generated B+tree (size ~20GB). It is not necessary to access the entire tree for finding the keys. Thus the use of page faults from the GPU avoids unnecessarily allocating physical memory for the entire tree. In Figure 8(b), we observe that XSBench allows savings in physical memory footprint during initialization, but memory footprint grows quickly as the entire allocated memory is gradually accessed over time. Deeper inspection into XSBench reveals that the biggest contributor to its memory footprint is the data structure for the nuclear energy grid. This energy grid is accessed to perform concurrent cross-sectional lookups on the GPU. We find that if a large number of lookups are performed (a parameter, default 15M) at random cross-sections, then eventually the entire energy grid is accessed. Thus the physical memory usage with and without GPU page faults converges. However, XSBench's physical memory footprint reduces substantially when a smaller number of lookups (e.g., 500K) are performed. We find that there is very little scope for consolidating physical memory for workloads like graph500 and CoMD. In graph500, the entire graph data structure is traversed and thus all of allocated memory is accessed. Similarly, for CoMD the entire allocated memory is needed by the GPU. Hence, the potential for reducing memory footprint varies across workloads and can be dependent upon the input.

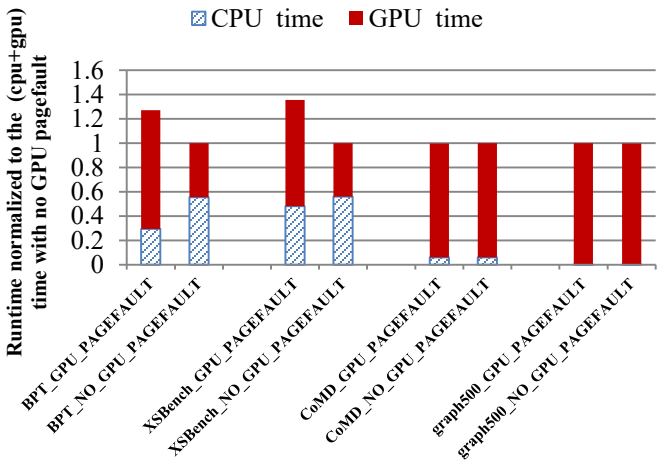

Figure 9. Performance overhead of GPU page faults.

Figure 9 shows the runtimes normalized to the runtimes with no page faults from the GPU. Each bar also shows a breakdown of runtime spent on the CPU and GPU. We observe that XSBench and BPT can incur significant performance degradations due to page faults from the GPU. We note that a larger fraction of the time is spent on the GPU if page faulting is used. This is expected as GPU page faults hinder concurrency in the GPU. Other workloads show little or no performance impact.

In summary, we find significant scope for research into heterogeneous software and hardware to reduce the page fault latency and enabling more concurrency in servicing them. These research efforts, however, should focus on enabling new capabilities in the runtime rather than improving the performance of legacy applications.

Observations and opportunities:

1. The latency to service a page fault from the GPU can be significantly higher than from the CPU.
2. Enhancements into system software to handle page faults synchronously can reduce this latency.
3. Software-hardware co-design is needed service a large number of concurrent faults from the GPU/accelerators.
4. It is imperative to scale CPU performance and resources to scale the GPU page fault servicing.
5. Future heterogeneous applications can reduce their physical memory footprints through the use of on-demand page faults from the GPU, although current applications may need to be re-written.

VI. Extending TLB Shootdowns to GPUs

We wrote a simple micro-benchmark that generates a large number of GPU TLB shootdowns. It generates both single-entry TLB invalidations and entire TLB flushes. Table 4 presents measurements and the breakdown of latency of a GPU TLB shootdown. The first column depicts the average latency of a GPU TLB shootdown. The first row shows the latency of

Table 4. GPU TLB shootdown analysis.

	Avg. GPU TLB shootdown latency (in nanosecs)	Breakdown of GPU TLB shootdown latency (in nanoseconds)			
		Initialization	IOMMU TLB invalidation	GPU TLB invalidation	Finalization
Single-entry shootdown	4234	79	1970	2021	162
All-entry shootdown	4409	122	2052	2035	299

invalidating a single entry, and the second row shows the latency of flushing the entire TLB. The average latency of a TLB shootdown is around 4.2-4.4 microseconds. This latency is comparable to typical times required to perform a TLB shootdown across 4 to 8 CPU cores [25]. We note that nearly an equal amount of time is spent in invalidating the IOMMU's and the GPU's TLB entries. We also note that there is no significant difference between the latency to invalidate a single entry or flushing the entire TLB.

In the Linux operating system, TLB shootdowns often take long time to complete for large systems with many nodes. This may potentially be a scaling bottleneck in the future. For example, future systems with multi-level memory [1] that migrate pages between different levels of memory will critically depend on TLB shootdown performance.

VII. Related Work

The advent of big-data workloads, often with poor access locality (e.g., graph processing algorithms, massive key value stores), have recently led to a surge of research on virtual memory for big-memory servers [3]–[8], [10], [13], [18], [19]. A number of research proposals have considered hardware and software mechanisms to improve the effective capacity of TLBs without additional area costs [4], [5], [8], [13], [18], [19], [19] with speculation [3], [20], and with approaches that better manage large pages [9], [15], [20]. In parallel, the emergence of the unified address space paradigm for APUs (and more broadly, heterogeneous systems) has prompted the first set of studies on GPU address translation and memory management units [21], [22]. For example Pichai, Hsu, and Bhattacharjee show that intelligent hardware page table walkers are crucial to the performance of throughput-oriented accelerators and are deeply tied to the operation of the wavefront scheduler for both round-robin (the default) and advanced dynamic wavefront formation strategies designed to improve cache locality and mitigate control-flow divergence overheads [21]. Similarly, Power, Hill, and Wood show that throughput-oriented, multithreaded page table walkers are critical for GPU performance, especially in conjunction with intelligently-designed translation caches [22]. Beyond these works on GPUs, researchers have begun considering address translation for fixed-function and programmable accelerators [23].

Unlike past works on GPU address translation, we are the first to characterize shared virtual memory behavior on a real

heterogeneous system with realistic workloads. Our work shows the benefits, challenges, and potentially interesting research avenues in this space by collecting results on the first generations of hardware and software that actually implement CPU-GPU shared address spaces. Our work sheds light on the detailed interactions in the address translation microarchitecture beyond the scope of prior works. As such, we believe that our study provides a foundation for guiding the research community on some of the most-pressing problems in the shared virtual memory paradigm.

VIII. Summary: Observations and opportunities

We summarize the lessons learned analyzing shared virtual memory in one of the first commercially available heterogenous processors. We discuss possible research opportunities in the space for future-generation heterogeneous processors.

Address translation: TLB misses in GPUs are currently an order of magnitue slower than that in CPUs. A GPU program's large memory-level parallelism, along with concurrent servicing of GPU TLB misses can potentially help to hide this latency. Research on techniques that increases TLB miss handling concurrency are crucial, particularly for throughput-oriented accelerators like GPUs. We observe that divergence in memory accesses impacts address translation more than the rest of the memory hierarchy. Although there has been a significant body of research in managing divergence in the cache hierarchy, there is a dearth of work that studies its impact on address translation. We find that prefetching translations usually aids performance, but under certain circumstances it can degrade performance. Research into software-hardware co-design for application-aware prefetechers for address transation will be useful. Finally, we find that large pages universally help in reducing address translation overheads. Hardware designers should build enhanced support for large pages and programmers should make use them. Furthermore, our preliminary experiments with second generation APU suggests that address translation is likely to become a bigger contributor to the performance overhead as the rest of the memory hierarchy is improved in future generation heterogenous processors.

Page fault: Dynamically allocating physical memory via page faults from the GPU could potentially enable significant memory consolidation for future applications with large footprints. However, we observed that servicing a GPU page fault on current systems can take an order of magnitude longer (3-82×) than that for CPU page faults. Addressing this challenge requires changes to both the hardware and software. Enhancements to both the system software and hardware to service a larger number of concurrent page faults could help mitigate the overheads in the page fault process.

TLB Shootdown: Our workloads encountered only a few instances of TLB shootdowns. TLB shootdown latencies for current heterogeneous systems are comparable to those in the CPU and are hence expensive. However, because TLB shootdowns are serialized, they could be a potential performance bottleneck in future systems, particularly for heterogeneous memory systems with frequent physical page migration. This is not an intrinsic limitation of the hardware, but architecting the OS to support concurrent shootdowns will be required as systems scale upward.

IX. Acknowledgements

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. The format PCIe® is a registered trademark of PCI-SIG Corporation. The format ARM® is a registered trademark of ARM Limited. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies. This work was supported in part by the National Science Foundation, under grant number 1337147.

X. References

[1] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. W. Wenisch, "Unlocking bandwidth for GPUs in CC-NUMA systems," presented at the International Symposium on High Performance Computer Architecture (HPCA), 2015.

[2] K. Atasu, F. Doerfler, J. van Lunteren, and C. Hagleitner, "Hardware-Accelerated Regular Expression Matching with Overlap Handling on IBM PowerEN Processor," in *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, 2013, pp. 1254–1265.

[3] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A Mechanism for Speculative Address Translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2011, pp. 307–318. Available: http://doi.acm.org/10.1145/2000064.2000101

[4] T. W. Barr, A. L. Cox, and S. Rixner, "Translation Caching: Skip, Don'T Walk (the Page Table)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2010, pp. 48–59. Available: http://doi.acm.org/10.1145/1815961.1815970

[5] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2013, pp. 237–248. Available: http://doi.acm.org/10.1145/2485922.2485943

[6] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating Two-dimensional Page Walks for Virtualized Systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2008, pp. 26–35. Available: http://doi.acm.org/10.1145/1346281.1346286

[7] A. Bhattacharjee, "Large-reach Memory Management Unit Caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2013, pp. 383–394. Available: http://doi.acm.org/10.1145/2540708.2540741

[8] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-level TLBs for Chip Multiprocessors," in

Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture, Washington, DC, USA, 2011, pp. 62–63. Available: http://dl.acm.org/citation.cfm?id=2014698.2014896

[9] Y. Du, M. Zhou, B. R. Childers, D. Mosse, and R. Melhem, "Supporting superpages in non-contiguous physical memory," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 223–234.

[10] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2014, pp. 178–189. Available: http://dx.doi.org/10.1109/MICRO.2014.37

[11] J. Goodacre, "The Evolution of the ARM Architecture Towards Big Data and the Data-centre (Abstract Only)," in *Proceedings of the 8th Workshop on Virtualization in High-Performance Cloud Computing*, New York, NY, USA, 2013, pp. 4:1–4:1. Available: http://doi.acm.org/10.1145/2535800.2535921

[12] Intel Quick Assist Technology, "Integrated Cryptographic and Compression Accelerators on Intel® Architecture Platforms." . Available: http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/integrated-cryptographic-compression-accelerators-brief.pdf

[13] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant Memory Mappings for Fast Access to Large Memories," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, New York, NY, USA, 2015, pp. 66–78. Available: http://doi.acm.org/10.1145/2749469.2749471

[14] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the Walkers: Accelerating Index Traversals for In-memory Databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2013, pp. 468–479. Available: http://doi.acm.org/10.1145/2540708.2540748

[15] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly TLB designs," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 210–222.

[16] H. K. T. Paul Stewart Crozier, "Improving performance via mini-applications," 2009.

[17] PCI Express, "Address Translation Services." . Available: http://composter.com.ua/documents/ats_r1.1_26Jan09.pdf

[18] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 558–567.

[19] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large-Reach TLBs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2012, pp. 258–269. Available: http://dx.doi.org/10.1109/MICRO.2012.32

[20] B. Pham, J. vesely, G. Loh, and A. Bhattacharjee, "Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have it Both Ways," presented at the IEEE/ACM International Symposium on Microarchitecture (MICRO), 2015.

[21] 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 *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2014, pp. 743–758. Available: http://doi.acm.org/10.1145/2541940.2541942

[22] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 568–578.

[23] Y. Sophia Shao, S. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Towards Cache-Friendly Hardware Accelerators," in *HPCA Sensors and Cloud Architectures Workshop (SCAW)*, 2015. Available: http://www.eecs.harvard.edu/~shao/papers/shao2015-scaw.pdf

[24] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, "CAPI: A Coherent Accelerator Processor Interface," *IBM J. Res. Dev.*, vol. 59, no. 1, pp. 7:1–7:7, Jan. 2015.

[25] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory," in *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011, pp. 340–349.

[26] "HSA Foundation." . Available: http://www.hsafoundation.com/

[27] "PCI Express." . Available: https://en.wikipedia.org/wiki/PCI_Express

.