Improving GPU Multi-tenancy with Page Walk Stealing

B. Pratheek* Neha Jawalkar* Arkaprava Basu
Department of Computer Science and Automation
Indian Institute of Science
{pratheekb, jawalkarp, arkapravab}@iisc.ac.in

***Abstract*—GPU (Graphics Processing Unit) architecture has evolved to accelerate parts of *a single application* at a time. Consequently, several aspects of its architecture, particularly the virtual memory, have embraced a *shared-mostly* design. This implicitly assumes that a single application and, thus, one address space is resident in the GPU at a time. However, recent trends, e.g., deployment of GPUs in the cloud, necessitate efficient multi-tenancy. Multi-tenancy is needed for sharing the physical resources of a large server-class GPU across multiple concurrent tenants (applications) for resource consolidation while ensuring fairness among the tenants.**

We first quantify how different components of GPU's virtual memory can impede multi-tenancy. We show that shared page walkers are a key bottleneck under multi-tenancy. We, therefore, propose *dynamic page walk stealing* that enables *soft* partitioning of the shared pool of walkers – reducing destructive interference between the tenants while also aggregating resources where possible. Over today's design, we improve throughput by 37%, and weighted IPC by 15%, on average, over 45 workloads.

***Index Terms*—graphics processing units, virtual memory, address translation, page table walkers, multi-programming**

I. Introduction

GPUs are key to accelerate a broad range of application domains with abundant data-level parallelism [33]. Traditionally, a GPU is tasked to accelerate portions of a single application at a time. This *"one-application-at-a-time"* usage model is reflected in the design of GPU's virtual memory. Typically, the entire GPU shares the last level TLB (here, L2) and a shared pool of page table walkers [45], [42], [43]. Shared TLBs and walkers are beneficial if the entire GPU executes kernel(s) from a single application (i.e., single address space) at a time – TLB entries and page walk requests can be shared across concurrent computations. This reduces the number of page table walks and, thus, the overheads of address translation.

In recent times, however, multi-tenancy has become a desirable feature on server-class GPUs. We define multi-tenancy as multiple applications concurrently executing on a shared GPU hardware as co-running tenants. As GPUs make their way into the cloud infrastructure [34], [21], [6] the ability to efficiently share a GPU across multiple concurrent tenants becomes a necessity. The sharing of resources is fundamental to cloud's economic model. Further, as GPU hardware resources continue to grow, kernel(s) from a single application may not be enough to keep an entire GPU busy [28], [30]. Multi-tenancy is thus becoming a desirable feature to ensure better utilization even outside of the cloud.

GPU vendors are evolving their software and hardware stack for catering to the needs of multi-tenancy. A good example is NVIDIA's Multi-process Service (MPS). Volta GPU architecture onward, MPS enables concurrent execution of kernels from different virtual address spaces in a GPU. This year, NVIDIA announced Multi-Instance GPU (MIG) technology in A100 GPUs [37]. MIG leverages SR-IOV [25] to expose multiple instances of a shared GPU to different tenants. It statically partitions compute, and memory resources among the instances for performance isolation [37]. AMD's MxGPU technology can similarly expose multiple instances of a GPU [1]. These evolving technologies demonstrate the growing need for better multi-tenancy support in GPUs.

Unfortunately, several key design choices in a typical GPU architecture implicitly assume the "one-application-at-a-time" usage model that runs contrary to the spirit of multi-tenancy. We focus on the virtual memory since the foremost requirement of multi-tenancy is the ability to concurrently service requests from multiple address spaces. However, the shared L2 TLB and the shared page walkers can induce uncontrolled interference across tenants resulting in significant performance loss. While static partitioning of resources can control interference it leads to severe resource under-utilization.

We quantify how various shared components of GPU's virtual memory impede multi-tenancy. While recent research has focused on the more obvious source of interference among tenants – the shared L2 TLB [5] – we find that the shared page walk subsystem is *also* a key source of performance degradation under multi-tenancy.

Independent page walk requests from co-running tenants can get interleaved with each other while they queue up at the walkers. Consequently, a tenant experiences latency due to page walks of another tenant(s). The impact of interleaving is particularly severe when one of the tenants generates page walks more frequently than the other. The former, in such cases, tends to keep all walkers busy with its requests for the most part. Consequently, when a walk request from other tenants arrives, it experiences high queuing latency as it gets queued behind many requests of the page-walk-intensive tenant (Section IV). This uncontrolled interference leads to significant loss of throughput. It can impair fairness too, where one of the tenants suffers disproportionate performance loss.

*Authors contributed equally.

2378-203X/21/$31.00 ©2021 IEEE
DOI 10.1109/HPCA51647.2021.00059

We, thus, design a multi-tenancy aware page walk system that limits uncontrolled interference to improve throughput and provides a way to tradeoff throughput with fairness.

We introduce the idea of *dynamic page (table) walk stealing* (DWS) – partially inspired by the work-stealing mechanisms in parallel language runtimes like Clik, TBB, etc. [14], [24]. We first partition the shared pool of walkers among the tenants. A walker in a partition serves page walk requests only from a given tenant (a.k.a., *owner*). This alleviates the interleaving of walk requests but leads to walker under-utilization and more unfairness. Every tenant does not generate walk requests at the same rate, and thus when walks from one tenant are queued up, the walkers owned by another tenant may go unused, leading to severe degradation in throughput.

To avoid walker under-utilization, DWS enables a page walker to *steal* a walk (work) from another walker owned by a different tenant. A walker can steal a walk *only if no* page walk request is pending from its owner tenant. Stealing improves utilization while strictly limiting interleaving of walks from co-running tenants. A walk from a tenant may have to wait for *at most* one walk from another tenant and on average, is a small fraction compared to tens in current designs (Section VII).

A subtle consequence of DWS's controlled sharing of walkers is a similarly controlled sharing of L2 TLB capacity. This is because allocations in the TLB by a tenant are directly proportional to the number of walks completed by it. Consequently, a larger (smaller) share of walkers leads to a larger (smaller) share of TLB capacity. Ultimately, DWS improves throughput by 55%, over the baseline for a subset of 32 (out of 45) virtual memory intensive workloads.

While DWS significantly improves throughput, we find that a better balance between performance and fairness is possible by judiciously loosening the condition for stealing. For example, DWS can be unfair to a tenant that requires servicing many page walks if it executes alongside another tenant that generates walk requests at a relatively low but steady rate. Stealing may not happen because other tenant's walkers, although less loaded, may not be free often enough.

An extension to DWS, called DWS++, thus, allows stealing if there is a vast difference in the walks queued from each tenant, even if there are non-zero number of walks pending. We observe that a page walk's impact on a tenant's performance varies based on its intensity of page walk generation. If a tenant generates many walks then quickly servicing one of its walks barely improves its performance since the queuing latency is already high. In contrast, if a tenant generates a moderate number of walks, then delaying its walks may cause a large slowdown. DWS++ thus, measures the rate of walk generation from co-running tenants at runtime and uses it, besides the difference in the number of walks queued by tenants, to decide whether to allow stealing. Importantly, DWS++ allows one to fine tune the balance between throughput and fairness by adjusting the aggressiveness of stealing.

In summary, we make the following contributions.

- We demonstrate that the uncontrolled sharing of page table walkers is a key reason behind the performance degradation under multi-tenancy in GPUs.
- We propose the idea of page walk stealing to limit uncontrolled interference among tenants for page walkers to significantly improve performance.
- We extend this to enable a trade-off between throughput and fairness by controlling when a walker steals.

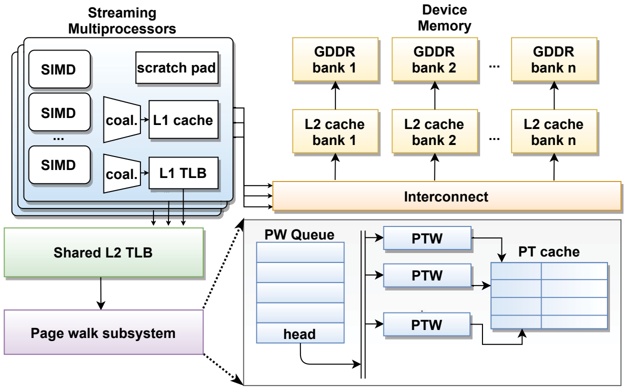


Fig. 1: Baseline GPU design

II. Background

Figure 1 depicts a typical GPU architecture. Streaming Multiprocessors (SMs) are GPU's basic computing blocks. There could be up to 64-80 SMs in a GPU. Each SM contains multiple Single-Instruction-Multiple-Data (SIMD) units, each with multiple *lanes* of execution (e.g., $16-32$). A SIMD unit executes a single instruction across all lanes in parallel. Each SM has a private L1 cache and a scratchpad that are shared across its SIMD units. When several data elements requested by a SIMD memory instruction falls on the same cache line, a hardware coalescer combines requests into single access to gain efficiency. The L1 cache is looked up after coalescing requests. All SMs share a larger L2 cache. The L2 cache is heavily banked, and each bank is typically connected to a DRAM channel. GPU's onboard memory (e.g., GDDR5/6 or HBM2) supports large bandwidth to service the needs of thousands of concurrent threads.

GPU virtual memory: Each SM has a private L1 TLB. In GPUs, often, memory accesses from a single SIMD memory instruction (load/store) falls on a single page (e.g., 4KB). These accesses are coalesced to a single address translation request before looking up the L1 TLB. On a L1 miss, a larger L2 TLB is looked up. Unlike L1 TLBs, the L2 TLB is shared by all SMs. On a L2 TLB miss, the translation request is sent downstream to the page walk system. The page walk system is shared by all SMs and is responsible for looking up the in-memory multi-level page table (cacheable). It houses multiple page table walkers (PTWs) that can concurrently service several walk requests (e.g., 8-16) [42], [43], [45]. When a new walk request arrives, all walkers could be busy servicing earlier requests. The request will then wait in a queue, called the page walk queue (PW queue), in the order it arrived at the page walk system [43]. When a walker finishes a walk, it returns the translation to the TLB and then picks the next request to service from the head of the PW queue.

While TLB hits are fast, page table walks can take hundreds of cycles. A walk requires up to four memory access to lookup a typical four-level page table. Therefore the page walk cache (PW cache) stores recently used *partial* translations [8], [43]. A partial translation helps skip accesses to upper levels of a page table's tree structure. Before starting the walk, the PW cache is looked up first for the longest prefix match on the virtual page number to be translated. On a hit, the number of required memory accesses reduces to 1-3, depending upon the length of the prefix match [8], [43].

Multi-tenancy in GPUs: While traditional GPU architecture is designed to serve memory requests from one tenant (application), multi-tenancy is becoming increasingly important. GPUs have made inroads into the public cloud infrastructures [21], [6] where resource consolidation and thus, efficient multi-tenancy is a necessity. GPUs with 64 SMs are commercially available and those with 128 SMs are around the corner [2], [46]. Keeping an entire GPU busy may not always be possible for a single tenant. Multi-tenancy is also the underpinning of effective virtualization where a GPU may need to be shared across multiple virtual machines.

It is possible to time-multiplex a GPU amongst tenants (temporal multi-tenancy) via preemption and context switching among tenants. However, temporal multi-tenancy does not necessarily address the issue of resource under-utilization [28], [30]. Due to a large number of threads and registers, context of a GPU program is much larger than that of a CPU program. Consequently, the overheads of preemption and context switching could outweigh the benefits.

We thus focus on spatial multi-tenancy where multiple tenants may concurrently execute on a shared GPU. Hereafter, by multi-tenancy, we refer to spatial multi-tenancy. An example of such a facility is NVIDIA's Multi-Process Service or MPS. With the Volta architecture onwards, MPS allows multiple tenants (virtual address spaces) to execute concurrently on a single GPU [35]. Subsets of SMs can be assigned to each tenant, while the memory subsystem is dynamically shared.

A common theme among these forms of spatial multi-tenancy is that while the SMs are partitioned among the tenants, the on-chip memory resources, including TLBs and caches, are dynamically shared and thus, may create uncontrolled contention. NVIDIA's upcoming A100 GPU promises to *statically* partition cache and memory bandwidth (though silent on TLBs/PTWs) [37]. However, we quantitatively find that static partitioning of virtual memory resources, such as PTWs, could significantly lower overall throughput (Section VII-D). In this work, we thus focus on how best to share GPU virtual memory resources under multi-tenancy.

III. Methodology

Before we delve into the quantitative analysis of contention in GPUs under multi-tenancy, we describe our methodology. We simulate the baseline GPU configurations and enhancements for DWS and DWS++ on the GPGPU-Sim simulator[7]. Specifically, we enhanced the MASK framework [5] that is derived from the MAFIA framework based off GPGPU-Sim [27]. Table I shows the baseline configuration. The simulator supports multi-tenancy and a comprehensive virtual memory system, including parallel page table walkers and a page walk cache. For most of this paper, we simulate two concurrent tenants without loss of generality. We further extend this to 3-4 tenants to demonstrate how the proposed design scales to a larger number of tenants in Section VII-F.

TABLE I: Baseline configuration

Shared multiprocessors	30 SMs, 1137 MHz, GTO scheduler
L1 TLB	Private, 32 TLB entries, 12 MSHR entries
L2 TLB	Shared, 1024 entries, 16 way set-associative
Page walkers	Shared, 16 walkers, 192 entry walk queue
Page walk cache	Shared across page walkers, 128 entries
L1 cache	Private to each SM, 16KB per SM
L2 cache	Shared, total 2MB, 16-way, 16 banks
Memory	16 channels, 345.6 MBps bandwidth

TABLE II: Workloads

Classification	**Benchmark**	**Description**
Light (MPMI <25)	MM	Matrix multiplication [44]
	HS	Temperature map of computer chips [15]
	RAY	Ray tracing [41]
	FFT	Fast Fourier Transform [44]
Medium (25 <MPMI <80)	LPS	Laplace solver for 3D grids [27]
	JPEG	JPEG encoding and decoding [27]
	LIB	Computing LIBOR swaption portfolio [27]
	SRAD	Speckle reducing an-isotropic diffusion [15]
	3DS	Updates elements of array as per pattern [27]
Heavy (MPMI >80)	BLK	Equation solver for market analysis. [27]
	QTC	Clustering with cluster size threshold [17]
	SAD	Sum of absolute differences [44]
	GUPS	Multi-threaded, random access [27]

Applications running as co-tenants do not necessarily have the same execution length (time). We thus continue simulation until both tenants have completed execution at least once. If one of the tenants finishes early then we relaunch the same application to ensure that the other tenant(s) continues to experience contention. We measure the IPC and other statistics for each tenant over all its completed executions. This follows the standard methodology employed in simulating multi-programmed workloads on CPUs [48], [26]

Workloads: Table II shows a set of applications drawn from the MAFIA framework [27] ordered by their TLB miss intensity. We quantify the TLB miss/page walk intensity of an application by its number of L2 TLB misses per million instructions (L2 TLB MPMI). An application is classified as Light (L), Medium (M), or Heavy (H) based on its L2 TLB MPMI. The workloads are then constructed by running these applications in pairs as co-running tenants.

The workloads (i.e., application/tenant pairs) can be classified as LL, ML, MM, HL, HM, or HH based on the classification of its constituents. There could be 78 such workloads. However, presenting individual data points for all those is neither possible nor necessary. Instead, we focus on 45 of them with representations from all the above-mentioned six possible workload classes. However, LL, ML, and MM workloads are not interesting since they are mostly agnostic to the virtual memory subsystem (quantified in the next section). Therefore,

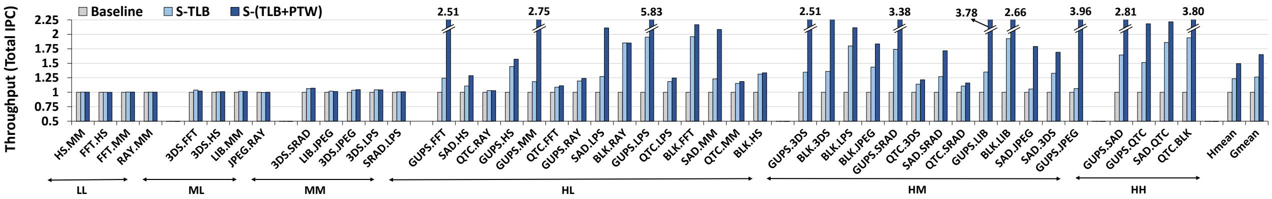


Fig. 2: Total IPC for baseline, separate TLBs and separate walkers. Normalized to baseline

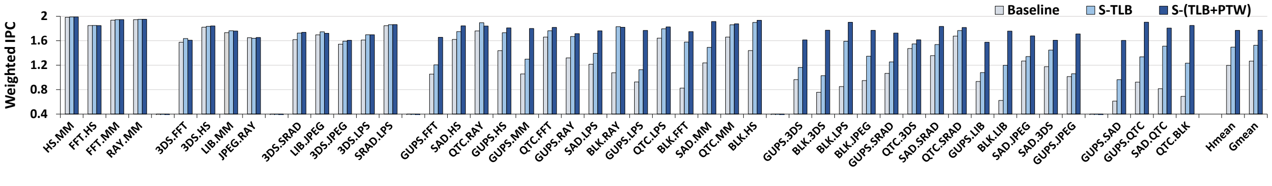


Fig. 3: Weighted IPC for baseline, separate TLBs and separate walkers

while we ensure that workloads from all six classes are represented, we focus more on the other three classes.

We observed that BLK, which has good cache locality, suffers many TLB misses. We find that this happens because the warp scheduler often schedules warps with different working sets together on an SM. This causes the TLB to thrash as it fails to keep disjoint working sets at the same time.

IV. Quantitative analysis of GPU multi-tenancy

A GPU's virtual memory (Figure 1) has two key sources of contention under multi-tenancy – the shared L2 TLB and the page walk system. These shared resources are beneficial when a GPU executes kernel(s) from one application (and thus, the same virtual address space) at a time. L2 TLB entries can be shared across SMs, and a page walk request from one SM may act as a prefetch for another. However, under multi-tenancy, tenants contend for the limited L2 TLB capacity and the shared pool of page walkers. Contention for the walkers also leads to contention in page walk caches.

To quantify performance headrooms from relieving contention at aforementioned shared virtual memory resources, we simulate three different configurations – ① Baseline, ② separate L2 TLB for each tenant (S-TLB), and ③ separate L2 TLB and page table walkers for each tenant (S-(TLB+PTW)). Baseline simulates the configuration in Table I.

In the S-TLB configuration, each tenant gets an exclusive copy of the L2 TLB (i.e., aggregate L2 TLB capacity doubles assuming two tenants). However, they still share the page walk subsystem. The performance difference between Baseline and S-TLB quantifies the performance that could be gained if the tenants did not interfere in the TLB.

Under S-(TLB+PTW), each tenant gets exclusive copies of both the L2 TLB and page table walkers (i.e., L2 TLB capacity and the number of walkers doubles). The performance difference between S-TLB and S-(TLB+PTW) quantifies the performance that could be further gained if we could also relieve contention in the page walkers. Note that *only* doubling the number of walkers is often useless and can distort analysis since it can cause thrashing in the TLB. A larger number of walkers allows walks to finish and fill the TLB at a higher rate, potentially thrashing the L2 TLB by evicting useful entries.

Figure 2 shows the throughput (normalized to baseline) of different workloads under the three configurations. We define the throughput as the total IPC ($\texttt{t\_IPC}^{\text{C}}$), which, for a given combination (C) of n tenants is $\sum_{i=1}^{n} IPC^{\text{C}}[i]$. For a cloud provider, throughput is indicative of the overall utilization of a GPU, and thus, the value that physical resource provides.

Each workload has three bars in the cluster representing the three configurations described earlier. Further, workloads are separated into groups based on the class a workload belongs to (marked on the graph). We observe that the throughput of different classes of workloads improves with S-TLB and S-(TLB+PTW) by different amounts, as expected. While there is little scope for performance improvement for LL, ML, MM workloads, there is significant scope for HL, HM and HH ones.

We observe that with S-(TLB+PTW), throughput increases by a significant margin over S-TLB. This increase is often higher than the throughput increase with S-TLB over the baseline. For example, the throughput over baseline improves by around 26%, on average, when the TLB is not shared (S-TLB). The throughput improves by a *further* 31% over S-TLB, if contention in the page walkers is avoided too (S-(TLB+PTW)). If we consider only the 32 workloads in the HL, HM and HH category, then throughput improves by an average of 38% with S-TLB, while S-(TLB+PTW) further improves it by 46%. Therefore, relieving contention in the page table walk system is key to efficient multi-tenancy.

While throughput indicates utilization, it does not reflect the relative slowdowns that tenants experience due to multi-tenancy. In a public cloud, relative slowdown, a.k.a., fairness amongst tenants, is important too. We use the weighted IPC, a standard metric to capture the IPC weighed by the relative slowdowns. The weighted IPC (w_IPC$^{\text{C}}$) for a given workload (C) with n concurrent tenants is calculated as $\sum_{i=1}^{n} \frac{IPC^{C}[i]}{IPC_{SA}[i]}$. IPC$_{\text{SA}}$[i] stands for the *stand-alone* IPC of a given tenant i in the combination. The stand-alone IPC is measured by executing that tenant *alone* on the baseline. A higher weighed IPC signifies that tenants experience relatively small perfor-

TABLE III: Interleaving of page walks

Workloads		Tenant 1	Tenant 2	Average
LL	HS.MM	0.01	0.012	0.011
	FFT.HS	0.01	0.032	0.021
	Arith. mean	0.017	0.043	0.03
ML	3DS.FFT	1.922	5.203	3.563
	LIB.MM	0.367	0.447	0.407
	Arith. mean	0.907	2.342	1.625
MM	3DS.SRAD	2.789	6.738	4.763
	LIB.JPEG	0.16	0.249	0.204
	Arith. mean	1.293	3.537	2.415
HL	BLK.HS	4.35	36.937	20.643
	GUPS.MM	22.843	109.697	66.27
	Arith. mean	12.236	72.346	42.291
HM	BLK.3DS	28.769	86.203	57.486
	GUPS.JPEG	7.233	131.907	69.57
	Arith. mean	15.282	72.761	44.021
HH	GUPS.SAD	72.539	77.208	74.874
	QTC.BLK	56.59	84.1	70.345
	Arith. mean	58.434	85.128	71.781

mance degradation under multi-tenancy. The possible values of weighted IPC range from zero to *n*.

Figure 3 shows the corresponding weighted IPCs. A significant improvement in weighted IPC could be gained by alleviating contention in the page walkers, more so than the shared TLB. For example, the weighted IPC increased by a further 16% when separate walkers were added on top of separate TLB. As in the previous figure, we observe that HL, HM and HH workloads show significant potential for improvement.

A. Interleaving of page walk requests

On further analysis of contention at the page walkers, we notice that unrelated walk requests from co-running tenants get *interleaved* while they queue up for servicing. Interleaving causes a tenant to wait for unrelated walk requests from another tenant, artificially increasing the walk latency. To quantify interleaving, we measured the average number of page walks of the other tenant that a walk request typically waits for. A higher number indicates larger interleaving.

Table III shows the measurements. Due to space constraints, we present only the average for each class of workloads and those of two randomly selected workloads in the class. The table has three data columns – the first column shows the average number of walks from the second tenant that a walk from the first tenant waits for. The second column shows the same for the walks from the second tenant. The last column shows the arithmetic mean of the two.

We make two observations. First, all classes of workloads, except LL, show a significant number of interleaved page walk requests (e.g., 42, on average, for HL workloads). More interleaving is observed for workloads with more page walk intensity. Second, one of the tenants typically experiences relatively more interleaving than the other within a given workload. This happens due to the relative difference in the page walk generation rate. When a walk request from the tenant that generates requests *relatively* infrequently arrives in the page walk queue, it is likely to get queued behind several other requests from the co-running tenant(s).

Does increasing TLB size and PTWs solve the problem? If there were unlimited hardware resources, there would never be any resource contention or interference. However, to extract the most out of the available resources and to ensure fairness, it is essential to avoid uncontrolled interference – something that DWS and DWS++ aim for. We experimented with doubling the TLB size and number of walkers in the baseline (2048-entry TLB, 32 PTWs) and found that it is around 8% slower than S-(TLB+PTW), which has the same resources but does not suffer from interference. This indicates that even with a large TLB and many PTWs, but with a relatively limited memory footprint of simulated workloads, interference is a key performance limiter. Section VII-E provides further evidence of the usefulness of limiting interference by demonstrating improvements with DWS for larger TLBs and more walkers. Further, increasing only the number of PTWs can lead to thrashing in L2 TLB, degrading performance. Increasing TLB size increases area, energy cost along with the lookup latency.

Summary: ① Contention in page walkers causes performance degradation in GPUs under multi-tenancy. ② This contention often manifests in the interleaving of unrelated walk requests from co-running tenants.

V. Dynamic page walk stealing

Our goal is to make GPU's virtual memory perform well under multi-tenancy. We aim to improve throughput while limiting unfairness among tenants. Driven by the analysis in previous section, we focus on the contention in the page walk system to achieve this goal. While contention for a shared resource is unavoidable in a resource-constrained environment, limiting the interleaving of requests can reduce its ill-effects.

A naive way to alleviate interleaving would be to partition the page walkers equally amongst tenants. A subset of walkers would service page walks from *only* a designated tenant (called the *owner*). However, this could lead to under-utilization of page walkers, and load imbalance under the common scenario when all tenants do *not* generate page walk requests at a similar rate. For example, if a tenant does not generate many page walks, then the set of walkers assigned to it may remain partially idle. However, if the other tenant is performance-sensitive to page walk latency, then it could have benefited from the idle walkers. Therefore, strict partitioning of walker lowers throughput (measured in Section VII).

To avoid under-utilization of page walkers while also limiting the interleaving of walks, we propose *dynamic walk stealing* (DWS). DWS starts by equally partitioning walkers among tenants. However, when a page walker finishes a walk, but no other walk request is pending from its owner tenant, the walker *steals* a pending (waiting) walk request from a different tenant (if one exists) for servicing. This simple strategy avoids underutilization of walkers and limits interleaving too. A page walk from a tenant may need to wait for *at most one* walk request from another tenant. Consequently, DWS *strictly limits* potential increase in page walk latency due to interleaving of walks from independent tenants by design.

DWS also has a secondary positive effect on the shared TLB. Since TLB entries are ultimately populated by results of walks, controlling the share of walkers among the tenants leads to similarly controlled sharing of L2 TLB entries. This helps limit the (potential) uncontrolled thrashing of one tenant's TLB entries by another (quantified in Section VII).

While DWS improves throughput by more than 37% on average (Section VII) over all 45 workloads, we find that a better balance between performance (throughput) and fairness is possible by loosening the condition for stealing. For example, DWS can be unfair to a tenant that generates many walk requests if it runs alongside a tenant that generates a steady but relatively infrequent stream of requests. It can introduce a large imbalance in the page walk queuing latencies across the tenants since a walker steals only if its owner has no pending walk. The imbalance in queuing can lead to a disproportionate slowdown experienced by one of tenants (i.e., unfairness).

DWS++, an enhancement to DWS, attempts to address this imbalance by allowing a walker to steal if there is a significant difference (above a threshold) in the number of walks queued by tenants. Stealing is allowed even when there are walks pending from the owner of a walker. Unfortunately, this re-introduces uncontrolled interleaving of walks from independent tenants. Therefore, DWS++ allows stealing only if there is a small number (below a threshold) of walks pending from the owner and ensures a walker does not continuously steal.

We further make a subtle observation that the impact of a single page walk on an application's (tenant) performance varies widely. Servicing *a* walk early from a tenant that generates many walks may barely speedup a tenant since the queuing delay is already high. In contrast, a tenant that generates a relatively moderate number of walks may slow down by a bigger margin if even a few of its walks are delayed. Therefore, it is important to consider the relative frequency of page walks from the tenants in allowing stealing.

DWS++, thus, allows a walker to steal more aggressively if the page walk generation rates of the tenants are *similar* (within a range). It makes it harder for a walker to steal a walk if the other tenant(s) is generating walks at a much higher rate than its owner. This avoids a scenario when one tenant speeds up little, but the other slows down significantly.

Further, DWS++ parameterizes the aggressiveness of stealing. By controlling the aggressiveness of stealing it is possible to strike different balances between throughput and fairness. This feature is useful in prioritizing fairness and QoS over performance based on the specific deployment requirement.

VI. Design and implementation

We here detail an implementation of DWS and DWS++ and discuss necessary hardware modifications. For ease of exposition, we assume two tenants and 8 page table walkers, but these are not fundamental to the design.

At a high level, implementing DWS and DWS++ requires taking actions during two events – ① when a new page walk request arrives at the page walk system, and ② when a walker finishes a page walk. When a new walk request arrives, it is sent to one of the page table walkers owned by the requesting tenant. When a walker becomes free after finishing a walk, it needs to decide whether to steal a walk or service a pending walk from its owner (if one exists). DWS steals only if no walk is pending from its owner, but DWS++ may decide to steal even otherwise. For this purpose, DWS++ needs to measure the relative frequency of arrival of walks from tenants during each *epoch* (defined in terms of a fixed number of walks) and the relative occupancy of page walk queues for tenants. These measurements are used in deciding whether to steal.

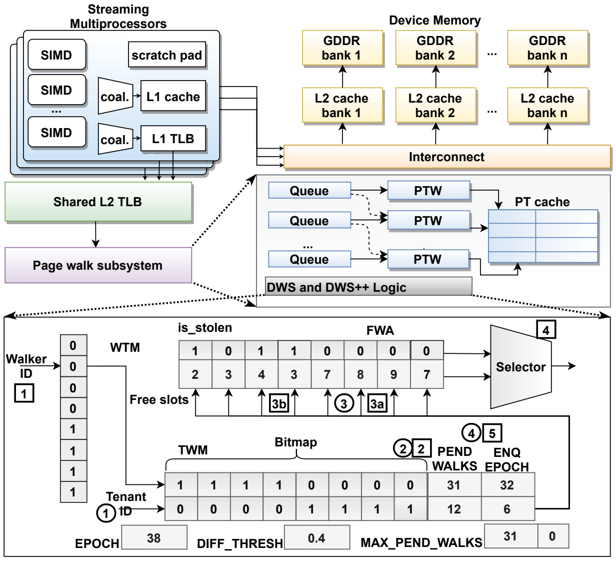


Fig. 4: Hardware for DWS and DWS++

A. Hardware modifications

We describe the necessary hardware modifications before detailing how they are used to implement DWS and DWS++. First, each address translation request is tagged with a unique tenant identifier (tenantID) for each actively running tenant. This is a single bit for two tenants but is typically small for a reasonable number of co-running tenants (e.g., 8). Next, as shown in Figure 4, the monolithic page walk queue for holding pending walk requests is divided equally into per-walker queues. The total number of queue entries remains unchanged. We also introduce an EPOCH counter (typically, 8 bits) that is incremented upon the arrival of a new walk request. When the counter reaches a predefined value (i.e., epoch length, default 200), the end of an epoch is signaled, and the counter itself is reset to zero.

We then add three small (10s of bits) direct-mapped structures (Figure 4). First, we keep an array of counters with one entry for every walker (8 in the figure, 16 in the default configuration). These counters track the number of free slots available in the corresponding walker's page walk request queue. We call this a Free Walker Array (`FWA`). `FWA` is a direct-mapped structure indexed by the page table walker ID (4 bits for 16 walkers). `FWA` also contains one bit per walker (is_stolen), which is set when the walker steals a walk and is reset whenever it services a walk from its owner (i.e., no stealing). We will later detail how DWS++ uses this bit and the epoch to control the aggressiveness of stealing.

Next, another direct-mapped table, called Tenant-to-Walker Mapping (TWM), tracks the subset of walkers owned by a tenant. It is indexed by tenantID (1-3 bits), and its entries have a bitmap of length equal to the total number of walkers (Figure 4). A bit in an entry's bitmap is set if the corresponding walker is owned by the given tenant. Each entry further keeps two more counters. The counter PEND_WALKS tracks the number of walk requests currently pending from the tenant. The counter ENQ_EPOCH tracks the number of walks that arrived from that tenant during the current epoch. These counters help to estimate walk queuing and the relative frequency of walks generation for each tenant.

The third direct-mapped table, called the walker-to-tenant mapping table (WTM), is indexed by a page table walker's ID (4 bits for 16 walkers), and each entry contains the tenantID the tenant that own the given walker. Besides, we keep a register called DIFF_THRES that is set dynamically to modulate the aggressiveness of stealing in DWS++.

Hardware state overhead: The total state overhead of new structures is only 192 bits. FWA contributes 80 bits, TWM contributes 80 bits, and WTM 32 bits. This assumes the default configuration with 16 page table walkers, two tenants, and a total of 192 entries to hold pending page walk requests from all tenants. For a larger number of tenants, the size of TWM grows linearly while that of WTM grows logarithmically.

B. *Operations of DWS and DWS++*

We first describe the operation of DWS by discussing actions taken upon the arrival of a new walk request, and when a walk finishes. We then detail how DWS++ extends them.

Initialization: We equally partition walkers among the co-running tenants and initialize the bitmap in TWM entries based on the assignment of the walkers to the tenants. All entries of the FWA array are initialized to the size of the individual walker queues (default, $12 = (192 \div 16)$). All counters are initialized to zero.

Arrival of a new walk request: When a walk request arrives at the page walk subsystem, it indexes into the TWM table using the tenantID of the request (Step ①). The bitmap in the TWM entry identifies the subset of walkers owned by the requesting tenant. It then looks up the corresponding counter values in the FWA array and chooses a walker with the highest value (Step ②). This corresponds to the walker that is least loaded. The walk request is then queued for the chosen walker, where it may wait before being serviced (Step ③). Finally, the PEND_WALKS counter in the tenant's TWM table entry is incremented and the counter in the FWA table corresponding to the chosen walker is decremented (Step ④).

Page walk completion: When a page table walker completes a walk, it first looks up its walk queue. If the queue is not empty, then it starts servicing the request at the head of the queue (Step $\boxed{\textbf{1}}$). Otherwise, it indexes into the WTM table with its walker ID to identify its owner tenant. It then looks up the corresponding PEND_WALKS counter by indexing into the TWM table with the owner's tenantID to check if there are pending page walks from the owner (Step $\boxed{\textbf{2}}$). If the counter is non-zero it consults the TWM to determine the walkers belonging to the owner. Thereafter, it consults the FWA entries of those walkers to select one with requests in its queue. The walker then starts servicing requests at the head of the chosen walker's queue. The FWA entry is updated accordingly (Step $\boxed{\textbf{3a}}$).

In Step $\boxed{\textbf{2}}$ above, if the value in PEND_WALKS was zero, then DWS *steals* a walk of another tenant (Step $\boxed{\textbf{3b}}$). It chooses a target tenant to steal from by finding a TWM entry with a non-zero value in its PEND_WALKS counter. The tenantID of the chosen entry is the target. Steps similar to those in $\boxed{\textbf{3a}}$ are then followed to choose one of the walkers owned by the target and dequeue a walk (*steal*) from its queue. Finally, the walker starts servicing the stolen walk (Step $\boxed{\textbf{4}}$).

In all cases, the PEND_WALKS counter corresponding to the tenant whose walk just finished is decremented. The entry in the FWA corresponding to that walker is incremented if it does not find a new request to service immediately.

Enhancement for DWS++: Under DWS++, a walker can steal even when there are walk requests pending from its owner. DWS++ first finds the difference in the number of walks queued by each tenant, and normalizes that by the number of entries in the page walk queue. Stealing is allowed if the normalized difference is greater than a threshold (DIFF_THRES). DWS++ controls the aggressiveness of stealing by dynamically changing the value of the threshold based on the relative frequency of page walk generation rate of tenants. As discussed earlier, DWS++ makes it harder (sets the threshold high) for a tenant's walker to steal if the other tenant is generating walk at a much lower rate than the walker's owner. DWS++ measures page walk generation rates by counting the number of walks from each tenant that arrive in an epoch (fixed number of walks). It *extends* the actions of DWS as follows.

When a new page walk request arrives, the global EPOCH counter and the per-tenant ENQ_EPOCH in the TWM are incremented. If the EPOCH counter reaches a predefined value (here, 200), the end of an epoch is signaled. When an epoch ends, the ratio of page walks enqueued by tenants is calculated by dividing the larger of the per-tenant ENQ_EPOCH values by the smaller one. If the ratio is small (e.g., $<= 1.5$) then DIFF_THRES is set to a smaller value (e.g., 0.4) to allow aggressive stealing. A larger ratio (e.g., $>= 2$) discourages stealing by setting the DIFF_THRES to a larger value (e.g., 0.8). The epoch counters are reset at the end of an epoch.

When a walk finishes, the walker first looks up its queue and checks if any walk is pending from the owner tenant as in DWS. If there are walks pending, DWS++ checks if the walker in consideration has just finished servicing a stolen walk by checking the is_stolen bit in the FWA table. If yes, it forbids stealing. This ensures that the interleaving of walks remains strictly bounded. If not, then it checks the occupancy of its own queues. If the queue occupancy is greater than a pre-defined threshold (QUEUE_THRES), stealing is forbidden. This ensures that a walker doesn't prioritize walks from another tenant when it itself has many walks pending. If both conditions are false, then DWS++ calculates the average occupancy of page walk queues for each tenant. It does so by computing the difference between the per-tenant PEND_WALKS counters in the TWM table and dividing it by the number of entries in the page walk queues. If this ratio is higher than the DIFF_THRES, then the walker steals walk in the same way as in DWS.

In summary, DWS and DWS++ require little additional hardware state (a couple of hundreds of bits), lookup of tiny direct-mapped structures, and perform some logic operations. Modifications are limited to the page walk system. We conservatively add latency for these operations in our evaluation. However, since all walk requests require at least one DRAM access and often suffer queuing delays, operations of DWS and DWS++ does not add any noticeable delay in the critical path.

C. Discussion

Scalability of DWS and DWS++: Hitherto, we assumed only two concurrent tenants. However, this is not fundamental to the design. Our design scales up to a larger number of concurrent tenants. However, the maximum needs to be fixed at design time. This constraint is in line with commercial multi-tenancy support. For example, NVIDIA's MPS supports up to a small number of concurrent tenants [35].

To support N concurrent tenants, the number of bits in tenantID and the number of entries in the TWM should be $\lceil \log_2 N \rceil$. Similarly, if the number of page table walkers is increased, only a few tens of bits of additional state would be required. There is barely any change in the operation of DWS. If it decides to steal, it can steal from the tenant with most pending walks or choose the target tenant randomly. For DWS++, the threshold after each epoch and the relative occupancy after each walk is calculated by first finding the tenant that has the most pending walks. The rest remains similar, whereby each tenant compares itself against that maximum. In Section VII-F, we quantitatively show the impact of scaling to more tenants.

Dynamically changing the number of tenants: We assumed apriori knowledge of the number of tenants. This may not be true in real execution. The number of tenants may increase dynamically on the arrival of a new tenant. It may decrease if a tenant finishes execution, but no other tenant is ready. Our design can accommodate these scenarios.

When a new tenant arrives, the contents of the TWM and WTM are updated. First, the allocation of walkers amongst the tenants must be recalculated and the bitmaps in the TWM entries are updated to reflect the new allocation. As before, we can equally partition them across all tenants, but this is not a necessity for DWS and DWS++. The entries of the WTM should also be updated accordingly.

When a new page walk request arrives, it will observe the updated contents of the TWM, and its requests will be routed to walkers accordingly. When a walker finishes walking, it will observe the updated contents of the WTM and TWM. There will be no disruption in servicing page walks, and the system will quickly converge to expected behavior.

If the number of tenants decreases, then we let the remaining tenants use freed walkers. As in the above, the partitioning of walkers across the tenants is re-calculated based on the new number of tenants. The contents of the TWM and WTM are then similarly updated to reflect altered execution conditions.

VII. Evaluation

We evaluate DWS and DWS++ to analyse their impact on ① throughput, ② fairness among the tenants, and ③ weighed IPC. Methodology is detailed in Section III. We perform a deep analysis of the impact of DWS and DWS++ on the interleaving of page walks, the average page walk latency and the frequency of walk stealing. We also quantitatively show how page walk stealing can subtly impact the contents of the shared TLB. We show how tuning DWS++'s parameters exposes a knob to choose different balances between the throughput and fairness. We measure the sensitivity to changing L2 TLB capacity and the number of walkers. Finally, we compare against relevant alternatives.

A. Performance and unfairness

Figure 5 shows the throughput (total IPC) normalized to the baseline (Table I). Each workload has three bars. The first bar represents baseline (always 1). The second bar represents total IPC with DWS, while the third represents DWS++. Table IV lists parameters for DWS++. It shows how the aggressiveness of stealing is tapered by adjusting DIFF_THRES with an increasing ratio of page walk generation rate among the tenants. Further, to avoid uncontrolled interleaving of walks, consecutive stealing by a walker is disallowed, and stealing happens only if no more than half of its queue entries are occupied (QUEUE_THRES). We empirically found that this strikes a good balance between the throughput and fairness.

From Figure 5, we observe that DWS, on average (geometric mean), across all 45 workloads, improves the total IPC by 37% over the baseline. If we focus on the subset of 32 workloads that are sensitive to virtual memory, then the improvement is 55%, on average. For example, when executing SAD and MM together or GUPS and JPEG together, the improvements are 1.86× and 3.8×, respectively. We, however, observe that for a couple of HH workloads, DWS is unhelpful. These workloads thrash the 1024-entry L2 TLB and DWS increases the thrashing by reducing page walk latency that, in turn, increases the TLB fill rate. Later in Figure 12, we will show that with a 2048-entry L2 TLB, DWS improves performance for those workloads too.

The speedup with DWS++ reduces slightly compared to DWS but still remains 34%, on average, over baseline. This

TABLE IV: Default DWS++ parameter values

Ratio of walk queued (R)	R <=1.5	1.5 <R <= 2	2 <R <= 3	3 <R <= 4	>4
DIFF_THRES	0.4	0.6	0.8	0.9	No stealing
QUEUE_THRES	0.51				

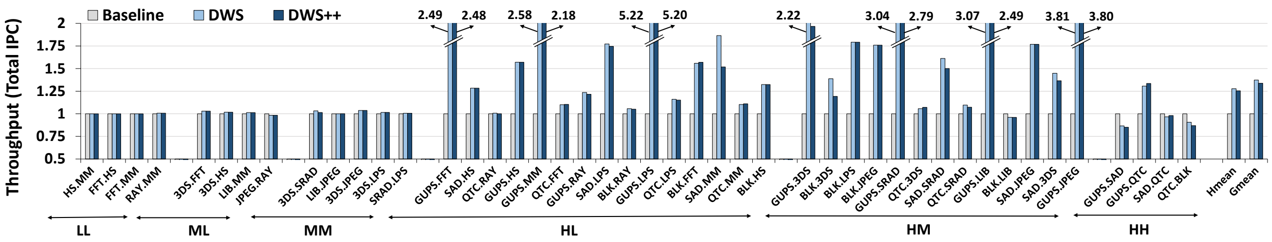


Fig. 5: Throughput (total IPC) for Baseline, DWS, and DWS++. Normalized to the Baseline

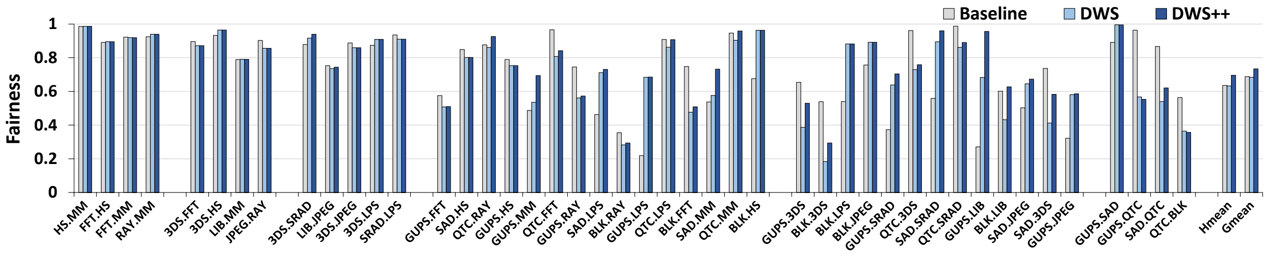


Fig. 6: Fairness in Baseline, DWS, and DWS++. Higher is better

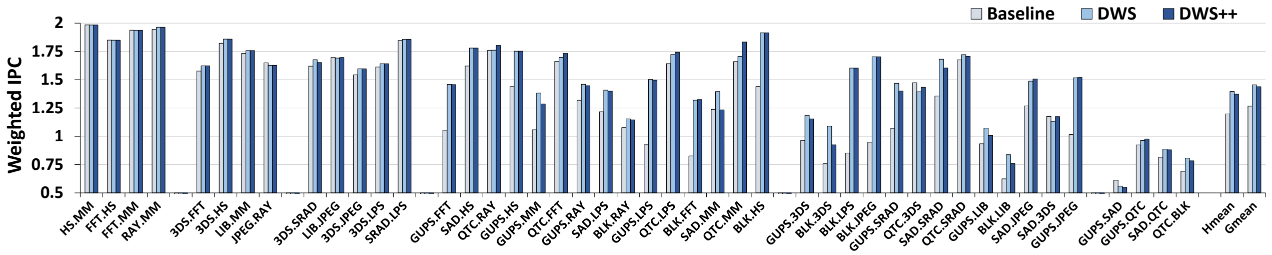


Fig. 7: Weighted IPC for Baseline, DWS, and DWS++

is expected; the purpose of DWS++ is to be able to moderate unfairness. Fairness often comes at a cost to throughput.

Next, we studied the fairness in execution. The fairness ($fairness_C$) for a given combination (C) of two concurrent applications is calculated as $fairness_c = \frac{min(S_A,S_B)}{max(S_A,S_B)}$ [18]. S_A represents the slowdown experienced by the tenant $\mathtt{A}$ while running along with tenant $\mathtt{B}$ and is calculated as $\frac{IPC^C[A]}{IPC_{SA}[A]}$. $IPC_{SA}[i]$ stands for the stand-alone IPC of a given tenant i. A value 1 signifies a completely fair execution, while 0 signifies a totally unfair execution. A higher value of this metric denotes better fairness and is desirable.

Figure 6 shows the fairness experienced by workloads under baseline, DWS and DWS++. DWS sometimes improves fairness over baseline but not always. For example, fairness in BLK.3DS drops with DWS. Even when executing stand-alone, BLK has a high demand for page table walkers. When executing alongside moderately page walk intensive 3DS, BLK is hamstrung by lack of walkers, and its IPC drops significantly. As walkers owned by 3DS are often not free, there is little opportunity for stealing walks. In contrast, 3DS, which was not getting enough page walkers in the baseline, experiences a large increase in IPC under DWS. Consequently, the relative slowdown experienced by each tenant increases, thereby hurting fairness. DWS++, as intended, improves fairness over DWS especially where DWS hurts fairness. For example, DWS++ improves fairness for BLK.3DS as BLK gets a relatively larger share of walkers due to aggressive stealing. Similarly DWS hurts fairness for GUPS.3DS, SAD.QTC, SAD.3DS, BLK.LIB and DWS++ moderates unfairness. On average over all workloads DWS++ provides best fairness.

Figure 7 shows the weighted IPC under the three configurations as before. Here, we observe that weighted IPC increases significantly under DWS compared to the baseline (on average, 15%). Improvement with DWS++ moderates slightly as it gives up some of the gains in the total IPC for fairness.

B. Analyzing DWS and DWS++

Interleaving: In Section IV-A, we demonstrated that the baseline suffers due to uncontrolled interleaving of page walk requests from independent tenants. Thus, a key objective of our design was to reduce interleaving. Table V presents the average number of page walk requests from the other tenant that a given walk request has to wait for (i.e., interleaving). Compared to tens in the baseline, average interleaving drops to a small fraction under both DWS and DWS++. Note that

TABLE V: Interleaving in Baseline, DWS, and DWS++

Workloads		Baseline	DWS	DWS++
LL	HS.MM	0.011	0.003	0.003
	FFT.HS	0.021	0.003	0.003
	Arith. mean	0.03	0.005	0.005
ML	3DS.FFT	3.563	0.162	0.162
	LIB.MM	0.407	0.032	0.032
	Arith. mean	1.625	0.071	0.071
MM	3DS.SRAD	4.763	0.096	0.097
	LIB.JPEG	0.204	0.014	0.013
	Arith. mean	2.415	0.113	0.113
HL	BLK.HS	20.643	0.379	0.379
	GUPS.MM	66.27	0.144	0.345
	Arith. mean	42.291	0.225	0.295
HM	BLK.3DS	57.486	0.056	0.456
	GUPS.JPEG	69.57	0.349	0.35
	Arith. mean	44.021	0.164	0.31
HH	GUPS.SAD	74.874	0.005	0.029
	QTC.BLK	70.345	0.019	0.084
	Arith. mean	71.781	0.016	0.056

TABLE VI: Percentage of page walks serviced by stealing

Workloads		DWS		DWS++	
		Tenant 1	Tenant 2	Tenant 1	Tenant 2
LL	HS.MM	0.21	0.19	0.21	0.19
	FFT.HS	1.01	0.03	1.01	0.03
	Arith. mean	0.81	0.35	0.81	0.35
ML	3DS.FFT	20.98	2.28	20.98	2.28
	LIB.MM	3.56	2.1	3.56	2.1
	Arith. mean	15.85	1.8	15.85	1.8
MM	3DS.SRAD	9.04	3.53	10.02	2.43
	LIB.JPEG	2.67	0.43	2.73	0.43
	Arith. mean	13.08	2.05	13.34	1.77
HL	BLK.HS	39.62	0.59	39.62	0.59
	GUPS.MM	29.1	0.15	31.22	0.31
	Arith. mean	34.53	0.78	35.2	0.82
HM	BLK.3DS	14.89	0.07	22.12	0.13
	GUPS.JPEG	40.55	0.04	40.73	0.05
	Arith. mean	26.29	1.09	30.06	1.1
HH	GUPS.SAD	3.34	0	3.6	0.21
	QTC.BLK	10.69	0.28	13.5	0.66
	Arith. mean	7.16	0.8	8.68	1.29

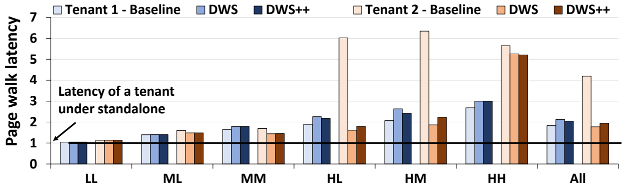
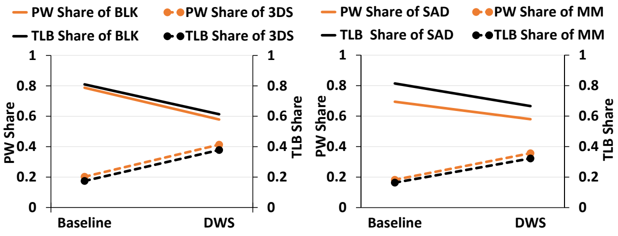


Fig. 8: Average walk latencies of different workload classes

interleaving is least for LL and HH workloads under DWS and DWS++. There is little need to steal for LL, and there is little scope to steal for HH. As expected, DWS++ slightly increases interleaving since it steals more aggressively.

Stealing: Table VI shows the percentage of walks from a tenant that are stolen by walkers owned by the other tenant. As expected, one of the tenants steals more than the other based on their relative page walk generation rate. Also, stealing happens more with DWS++ than with DWS.



(a) 3DS and BLK

(b) SAD and MM

Fig. 9: Effect of page walker share on the L2 TLB content

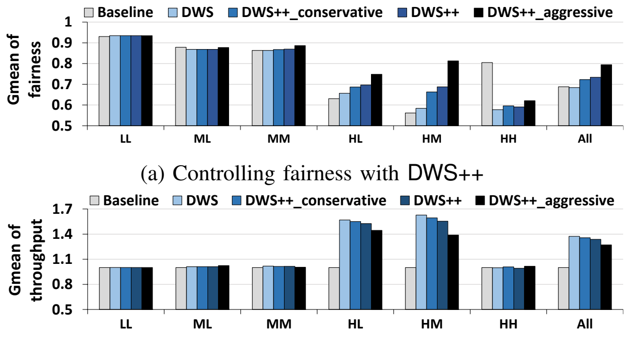
Page walk latency: Under multi-tenancy, walk latency experienced by a tenant can increase due to resource contention and interleaving of walks from co-running tenant(s). While some amount of queuing in a resource-constrained environment is unavoidable, partitioning and stealing of page walkers can ameliorate artificial increase in the latency due to interleaving.

Figure 8 shows how average walk latency for constituent tenants in workloads of different classes change under baseline, DWS and DWS++. For each class of workloads, there is one cluster of bars for one of the tenants in the workload. Each bar represents the average (gmean) of normalized walk latency experienced by the tenant under the given configuration. Walk latency is normalized to the latency experienced by the tenant when executing alone. Thus, the height of a bar depicts how the walk latency experienced by a co-running tenant increased due to multi-tenancy under various configurations.

For HL, HM, and HH workloads, the walk latency of the tenants in the baseline (here, Tenant-2) jumps 5-6× over the stand-alone walk latency for that tenant. The relatively less page-walk-intensive tenant is starved of walkers due to the uncontrolled interleaving of walks and resource contention by other tenant. Under DWS, no tenant is starved of walkers and the walk latency rationalizes. DWS's partitioning of walkers helps. At the same time, due to opportunistic stealing, walk latency of relatively page-walk intensive does *not* increase by the same margin, leading to overall throughput improvement. As expected, DWS++ moderates the relative impact in walk latency of the tenants in order to improve fairness. DWS and DWS++ have limited impact on HH workloads though. HH workloads are entirely resource-constrained. Unless more hardware resources (TLBs, walkers) are employed, there is little scope of improvement by reducing the interleaving alone.

We note that improvements due to DWS are limited when neither of the applications stresses the TLB enough (i.e., both applications in L or M category). This is expected; if there are not many page walk requests generated by the applications, then there is little scope for improvement. However, the improvements are most significant when one of the applications has a high TLB miss rate (H-category). DWS's smart management of page walkers among the applications yield high dividends in such cases. However, when both applications thrash the TLBs, better resource management can do little to help performance unless more hardware resources, i.e., large TLBs and/or more walkers, are deployed.

Impact of stealing on TLB: We observe that by controlling



(a) Controlling fairness with DWS++

(b) Impact on throughput

Fig. 10: Balancing fairness and throughput with DWS++

TABLE VII: Parameters for DWS++ variants

Ratio of walks queued(R)	DIFF_THRES		
	Conservative	DWS++	Aggressive
R <= 1.5	0.4	0.4	0.3
1.5 < R <= 2.0	0.6	0.6	0.3
2.0 < R <= 3.0	0.8	0.8	0.3
3.0 < R <= 4.0	0.9	0.9	0.3
4.0 < R	No stealing	No stealing	0.3

QUEUE_THRES	Conservative	DWS++	Aggressive
	0.17	0.51	0.51

the share of page walkers, DWS and DWS++ also impact the share of L2 TLB capacity of each tenant. TLB entries are filled with results of walks, and thus, if a tenant receives a relatively more (less) share of walkers, it also gets a bigger (smaller) share of TLB capacity.

Figure 9 captures this effect for the two representative workloads (subfigures). We calculate the average fraction of all walkers that are busy servicing walks for each tenant in a workload and the average fraction of L2 TLB entries occupied by each tenant. We plot these on the left and right y-axes in subfigures, respectively. The left half of each subgraph represents the baseline and the right half DWS. In each half, there are four data points for TLB and page walker share for each constituent tenant. We then join the data points for corresponding page walker and TLB share in baseline and DWS for ease of exposition. For both workloads, we observe that as DWS alters the share of walkers that a tenant gets, it also proportionally alters the share of TLB entries. This shows that DWS is able to control the TLB share among tenants, as well. The same is true for DWS++ too (not shown).

C. Balancing fairness and throughput with DWS++

A key feature of DWS++ is that it can balance throughput and fairness by controlling the aggressiveness of stealing. By altering the parameters of DWS++ (Table VII) we created two more configurations. In one configuration stealing conservative than the default (Table IV) and while the other steals more aggressively. Figure 10a shows how fairness can be controlled by these configurations. We observe that it is possible to achieve better fairness through aggressive stealing. As expected, however, throughput drops slightly with aggressive stealing (Figure 10b). We also note that for HH workloads, the fairness of DWS and DWS++ are lower than the baseline. Both tenants are page-walk intensive here and stealing by walker of a page-walk intensive tenant hurts it more than the improvement in other tenant's performance.

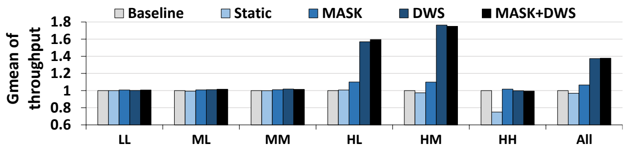


Fig. 11: Comparison with alternatives

D. Comparison with alternatives

We compare DWS with the naive static partitioning of walkers and with a recent work called MASK [5]. The static partitioning, like DWS, equally partitions the walkers amongst the tenants, but it disallows stealing. A comparison with static partitioning demonstrates the need for stealing. MASK, like DWS and DWS++, aims to improve the GPU virtual memory under multi-tenancy. However, it focuses on contention in the L2 TLB and contention between data and page table entries in the caches, unlike our work on addressing contention in the walkers. MASK employs token-based flow control and utility-based cache bypassing for PTEs. To simulate MASK, we used the source code made available by the authors [3].

Figure 11 shows the throughput of baseline, static partitioning, MASK, DWS and MASK+DWS. Since MASK and DWS are orthogonal, MASK+DWS employs both together. The height of each bar is normalized to the baseline. We show the measurements for each class of workloads only due to space constraints. First, we observe that static partitioning degrades performance over baseline. Thus, stealing is key. Second, DWS outperforms MASK by 29%, on average. We note that MASK's improvement over baseline is muted compared to that in the original article [5], although we used the author's source code. Our baseline, unlike in [5], includes a page walk cache and larger L2 TLB (1024 vs. 512 entries). Importantly, authors simulated *unlimited* number of walkers. We suspect that a more realistic baseline limited the benefits of MASK. In short, we demonstrate that DWS can work alongside MASK but may not bring significant additional benefit over running alone.

E. Sensitivity studies

Figure 12 shows the sensitivity of DWS to changes in the number of walkers and the L2 TLB capacity. There are two clusters of bars for each workload class. In each cluster, there are three bars representing a varying number of L2 TLB entries (first cluster) and a varying number of walkers (second cluster). The height of each bar represents the geometric mean of the improvement of that class of workloads, over the baseline with the same number of TLB entries or walkers as specified in the legend. For example, the bar for 12 walkers shows DWS's improvement over baseline, both executed with 12 walkers. All other configurations remain unchanged.

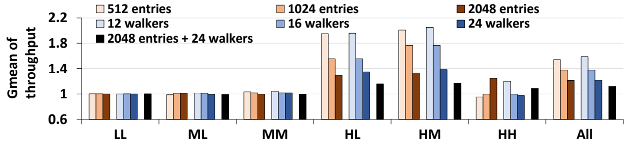


Fig. 12: Sensitivity of DWS to the number of page walkers and the number of entries in the L2 TLB. Throughput normalized to the corresponding baseline

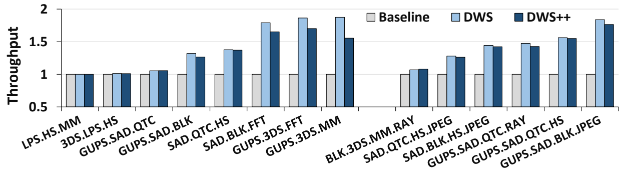


Fig. 13: Throughput with three and four tenants

As expected, with more walkers, the improvement from DWS moderates, but still remains substantial, particularly for HL and HM workloads. Similarly, a bigger TLB or both bigger TLB and more PTW reduce improvements with DWS, but they still remain substantial. The only exception is HH workloads. There, the larger capacity reduces thrashing in TLB, making DWS more effective. This is more prominent when both the number of walkers and TLB size increases.

F. Scaling beyond two tenants

As discussed earlier, DWS and DWS++ are scalable beyond two tenants. We simulated a few workloads with three and four tenants. These simulations take very long to finish and are impractical to run a large number of workloads. However, they allow us to sample the scalability of DWS and DWS++. We kept the TLB size same as baseline and slightly changed the number of walkers to allow equal and non-fractional number of walkers per tenant.

Figure 13 shows the throughput under baseline, DWS, DWS++ for a few randomly selected combinations of tenants. The height of each bar is normalized to the baseline. We observe that DWS is able to provide significant increase in throughput even with more than two concurrent tenants (up to $1.9\times$ and more than $1.25\times$ in 10 out of 14 workloads).

G. Performance of DWS with large pages

GPUs can support large page sizes such as 64KB and 2MB [36]. To demonstrate DWS's usefulness even in the presence of large pages, we simulated a few workloads with enhanced memory footprint that ran for weeks. Figure 14 shows normalized throughput with DWS over baseline when using 64KB pages. We observe that DWS improves performance even in the presence of large pages. In general, DWS enables a better utilization of hardware resources for address translation, which is important even when large pages are used.

VIII. Related work

Jog et al. studied interference in the GPU memory under multi-tenancy and proposed a memory scheduler that differentiates among accesses from different tenants [27]. Prior works explored how to execute multiple kernels concurrently on a GPU to increase utilization [38], [47]. Researchers also explored how to design and connect large multi-socket GPUs [50], [32]. These works lay the foundation for designing bigger GPUs but do not focus on the virtual memory.

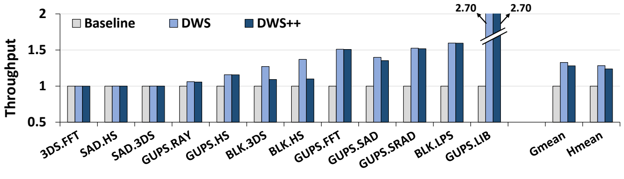


Fig. 14: Effect of large pages on throughput

GPU's virtual memory under single-tenancy is studied widely. Lowe-Power et al. demonstrated the importance of coalescer, shared L2 TLB, and multiple page walkers [31]. These are part of our baseline. Pichai et al. proposed TLB-aware warp scheduler to reduce TLB misses [40]. Vesely et al. showed how memory access divergence can increase address translation overheads [45]. Cong et al. proposed to use CPU's page walkers for GPUs [22]. Yoon et al. proposed the use of virtual caches in the GPU to defer address translation [49]. This approach removes address translation from the critical path but makes supporting multi-tenancy harder. Haria et al. leveraged identity mapping between virtual and physical memory to avoid page walking [23]. Shin et al. demonstrated that reordering of address translation requests [42] and coalescing of concurrent page walks can aid performance [43]. Ganguly et al. studied a tree-based prefetching mechanism for unified virtual memory and the page eviction algorithm [20]. In Mosaic [4], Rachata et. al. explored the trade-offs between large pages and overheads of demand paging. However, unlike ours, these works do not explore multi-tenancy.

CPU's virtual memory management is a well-researched topic. Bhattacharjee et al. proposed inter-core cooperative TLB prefetchers for multi-threaded workloads [13]. Pham et al. utilized intermediate contiguity to map multiple pages using a single TLB entry [39]. MIX TLBs showed how to efficiently support multiple page sizes in a single TLB [16]. SpecTLB predicts address mappings to hide the TLB miss latency [9]. Multiple proposals explored the use of segments to selectively bypass page walks [10], [11], [12], [19], [29]. While lessons from these works are useful, they are not directly applicable to GPU.

IX. Conclusion

As GPUs seep into the cloud, multi-tenancy assumes importance. We show that contention at the page walkers causes performance loss in GPUs under multi tenancy. We proposed DWS to employ walk stealing to reduce interference in page walks from concurrent tenants while also ensuring high walker utilization. DWS improve the throughput by over 37%, on average. Further, by controlling the aggressiveness of stealing, DWS++ can balance performance and fairness.

X. Acknowledgement

We thank anonymous reviewers of HPCA 2021 for their thoughtful review of this work. We thank Ashish Panwar, Ravi S. Anupindi, and Shweta Pandey for their feedback on the drafts of this article. This work is supported by Pratiksha Trust, Bangalore, and by a research grant from VMware Inc..

References

[1] AMD, "AMD MxGPU," https://www.amd.com/en/graphics/workstation-virtual-graphics.

[2] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "Mcm-gpu: Multi-chip-module gpus for continued performance scalability," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 320–332. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080231

[3] R. Ausavarungnirun, "Source code of MASK," https://github.com/CMU-SAFARI/Mosaic, 2018.

[4] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: A gpu memory manager with application-transparent support for multiple page sizes," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 136–150. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123975

[5] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 503–518. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173169

[6] A. AWS, "P3 instances with v100," 2020. [Online]. Available: https://aws.amazon.com/ec2/instance-types/p3/

[7] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, April 2009, pp. 163–174.

[8] 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*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 48–59. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815970

[9] 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*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 48–59. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815970

[10] 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*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 237–248. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485943

[11] Basu, Arkaprava, "Revisiting virtual memory," 2013, http://research.cs.wisc.edu/multifacet/theses/arka_basu_phd.pdf.

[12] Basu, Arkaprava and Hill, Mark D. amd Swift, Michael M., "Virtual memory management system with reduced latency," 2015, https://patents.google.com/patent/US9158704B2/en.

[13] A. Bhattacharjee and M. Martonosi, "Inter-core cooperative TLB for chip multiprocessors," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 359–370. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736060

[14] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. New York, NY, USA: ACM, 1995, pp. 207–216. [Online]. Available: http://doi.acm.org/10.1145/209936.209958

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797

[16] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 435–448. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037704

[17] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735702

[18] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

[19] 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*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 178–189. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.37

[20] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 224–235. [Online]. Available: https://doi.org/10.1145/3307650.3322224

[21] Google, "Cloud gpus," 2019. [Online]. Available: https://cloud.google.com/gpu/

[22] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting address translation for accelerator-centric architectures," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 37–48.

[23] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing memory in heterogeneous systems," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 637–650. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173194

[24] Intel, "Intel thread building blocks," https://software.intel.com/en-us/tbb, accessed: 2019-11-25.

[25] Intel, "Pci-sig sr-iov primer," https://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html.

[26] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 208–219. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454145

[27] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of gpu memory system for multi-application execution," in *Proceedings of the 2015 International Symposium on Memory Systems*, ser. MEMSYS '15. New York, NY, USA: ACM, 2015, pp. 223–234. [Online]. Available: http://doi.acm.org/10.1145/2818950.2818979

[28] Jog, Adwait, "Design and analysis of scheduling techniques for throughput processors," 2015, https://etda.libraries.psu.edu/catalog/26480.

[29] 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*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 66–78. [Online]. Available: http://doi.acm.org/10.1145/2749469.2749471

[30] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing gpu concurrency in heterogeneous architectures," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 114–126. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.62

[31] J. Lowe-Power, M. Hill, and D. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *Proceedings of International Symposium on High-Performance Computer Architecture*, ser. HPCA '14, 02 2014, pp. 568–578.

[32] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans, "Beyond the socket: Numa-aware gpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 123–135. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124534

[33] NVIDIA, "GPU-accelerated applications," 2016, http://images.nvidia.com/content/tesla/pdf/Apps-Catalog-March-2016.pdf.

[34] Nvidia, "Gpus everywhere," 2019. [Online]. Available: https://blogs.nvidia.com/blog/2017/05/08/microsoft-azure-gpu-instances/

[35] NVIDIA, "Nvidia multi-process service," 2019, https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[36] Nvidia, "Nvidia pascal mmu," 2019, https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf.

[37] NVIDIA, "Nvidia multi-instance gpu user guide," 2020, https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html. [Online]. Available: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html

[38] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving gpgpu concurrency with elastic kernels," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2013, pp. 407–418.

[39] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced large-reach TLBs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2012, pp. 258–269.

[40] 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*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 743–758. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541942

[41] M. Shih, Y.-F. Chiu, Y.-C. Chen, and C.-F. Chang, "Real-time ray tracing with cuda," in *Algorithms and Architectures for Parallel Processing*, A. Hua and S.-L. Chang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 327–337.

[42] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu, "Scheduling page table walks for irregular gpu applications," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 180–192.

[43] S. Shin, M. LeBeane, Y. Solihin, and A. Basu, "Neighborhood-aware address translation for irregular gpu applications," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. Piscataway, NJ, USA: IEEE Press, 2018, pp. 352–363. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00036

[44] J. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. Liu, and W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[45] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee, "Observations and opportunities in architecting shared virtual memory for heterogeneous systems," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 161–171.

[46] T. Vijayaraghavan, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, O. Kayiran, M. Meswani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan, "Design and analysis of an apu for exascale computing," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 85–96.

[47] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel: Fine-grained sharing of gpus," *IEEE Comput. Archit. Lett.*, vol. 15, no. 2, pp. 113–116, Jul. 2016. [Online]. Available: https://doi.org/10.1109/LCA.2015.2477405

[48] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 174–183. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555778

[49] H. Yoon, J. Lowe-Power, and G. S. Sohi, "Filtering translation bandwidth with virtual caching," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 113–127. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173195

[50] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining hw/sw mechanisms to improve numa performance of multi-gpu systems," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. Piscataway, NJ, USA: IEEE Press, 2018, pp. 339–351. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00035