DUCATI: High-performance Address Translation by Extending TLB Reach of GPU-accelerated Systems

AAMER JALEEL, EIMAN EBRAHIMI, and SAM DUNCAN, NVIDIA

Conventional on-chip TLB hierarchies are unable to fully cover the growing application working-set sizes. To make things worse, Last-Level TLB (LLT) misses require multiple accesses to the page table even with the use of page walk caches. Consequently, LLT misses incur long address translation latency and hurt performance. This article proposes two low-overhead hardware mechanisms for reducing the frequency and penalty of on-die LLT misses. The first, *Unified CAche and TLB (UCAT)*, enables the conventional on-die Last-Level Cache to store cache lines and TLB entries in a single unified structure and increases on-die TLB capacity significantly. The second, *DRAM-TLB*, memoizes virtual to physical address translations in DRAM and reduces LLT miss penalty when UCAT is unable to fully cover total application working-set. DRAM-TLB serves as the next larger level in the TLB hierarchy that significantly increases TLB coverage relative to on-chip TLBs. The combination of these two mechanisms, *DUCATI*, is an address translation architecture that improves GPU performance by 81% (up to 4.5×) while requiring minimal changes to the existing system design. We show that DUCATI is within 20%, 5%, and 2% the performance of a perfect LLT system when using 4KB, 64KB, and 2MB pages, respectively.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Hardware** → **Semiconductor memory**; *Emerging technologies*; *Emerging interfaces*; • **Software and its engineering** → **Memory management**;

Additional Key Words and Phrases: GPU, virtual memory, TLB, caches, high bandwidth memory

ACM Reference format:
Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. 2019. DUCATI: High-performance Address Translation by Extending TLB Reach of GPU-accelerated Systems. *ACM Trans. Archit. Code Optim.* 16, 1, Article 6 (March 2019), 24 pages.
https://doi.org/10.1145/3309710

1 INTRODUCTION

Heterogeneous systems composed of latency optimized cores (e.g., CPUs) and throughput optimized cores (e.g., GPUs, MIC) (Duran and Klemm 2012) are becoming the defacto organization of future high-performance computing platforms. Such systems typically consist of a hybrid memory system (Jeffers et al. 2016; Macri 2015; GP100 2016) that is composed of commodity DRAM (JEDEC 2013a) and stacked DRAM (JEDEC 2013b; hmc 2013). Furthermore, such systems also support *Shared Virtual Memory* (HSA Foundation 2014; Negrut et al. 2014) where both CPUs and GPUs access the entire hybrid memory address space using a unified virtual address space.

Authors' addresses: A. Jaleel, 392 Hudson St, Northborough, MA 01532; email: ajaleel@nvidia.com; E. Ebrahimi, 101 Colorado St Apt 1906, Austin TX 78701; email: eebrahimi@nvidia.com; S. H. Duncan, 37 Wollaston Ave, Arlington, MA 02476; email: sduncan@nvidia.com.

1544-3566/2019/03-ART6
https://doi.org/10.1145/3309710

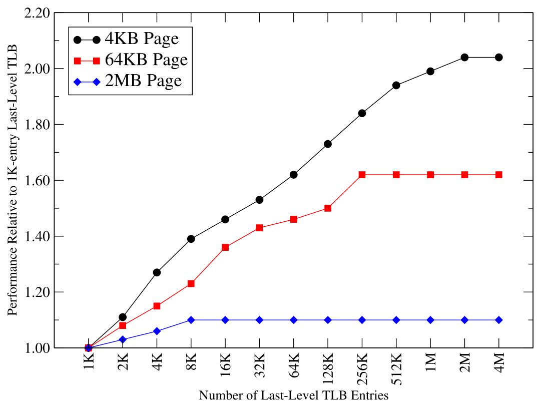


Fig. 1. Performance sensitivity to LLT entries.

In the virtual memory framework, depending on the page table implementation, address translation requires one or more page table accesses (Bhargava et al. 2008). To avoid the long memory access latency, processor architects cache recent address translations using an on-chip multi-level translation look-aside buffer (TLB) hierarchy. Growing application working-set sizes continue to stress the on-chip *Last-Level TLB (LLT)* (Barr et al. 2011; Basu et al. 2013; Bhattacharjee et al. 2011; Pham et al. 2012). LLT misses are latency sensitive operations that require one or more serial accesses to the page table. Therefore, virtual memory performance is dependent on the performance of the LLT. Reducing LLT miss latency enables instructions depending on the missing TLB entry to make faster forward progress.

A recent real system measurement study shows significant opportunity to improve shared virtual memory performance of heterogeneous CPU-GPU systems (Vesely et al. 2016). Specifically, they show that LLT misses are an order of magnitude slower on the GPU relative to the CPU. Thus, we focus on improving GPU LLT miss overhead in CPU-GPU systems by reducing both LLT miss *frequency* and LLT miss *latency*.

A simple solution to reduce LLT miss frequency would be to increase LLT size to cover the application working-set size. Figure 1 shows GPU performance sensitivity to LLT size relative to a 1,024-entry shared LLT.[1] The x-axis shows TLB sizes, while the y-axis shows average performance relative to the baseline for three different page sizes: 4KB, 64KB, and 2MB. The figure shows that performance is highly sensitive to the number of LLT entries with smaller page sizes. On average, a two million entry LLT improves performance by 2× with a 4KB page size. Similarly, a 256k-entry LLT improves performance by 60% with 64KB page size. Finally, an 8k-entry LLT improves performance by 10% with 2MB page size. Note that as application memory working-set sizes grow larger, the performance sensitivity to larger LLT size also increases with large page sizes.

Unfortunately, on-die area limitations prohibit simply increasing the LLT size, especially with smaller page sizes. Consequently, prior work has investigated improving performance of virtual memory translation using large pages to help TLB coverage. However, large pages can create unintended OS performance overheads (Talluri and Hill 1994; Talluri et al. 1992) due to memory imbalance (Gaud et al. 2014), memory fragmentation, paging (Chou et al. 2014), page creation, and page splitting (Pham et al. 2015) in emerging Non-Uniform Memory Access (NUMA) GPU systems (Young et al. 2018b). As such, there is a growing need to improve LLT coverage without relying on large pages alone.

[1]Section 3 discusses workloads and simulation methodology.

We propose hardware mechanisms to improve LLT coverage and LLT miss penalty on emerging GPU systems. Our first mechanism, *Unified Cache and TLB (UCAT)*, reduces the frequency of on-die LLT misses by enabling the conventional unified Last-Level Cache (LLC) to also hold TLB entries. UCAT augments the existing on-die LLT and increases on-die TLB coverage by potentially allowing as many TLB entries as there are cache lines in the conventional on-chip LLC.

However, UCAT *does not* reduce the LLT miss penalty incurred from walking the page table. This is because an LLT miss still requires multiple long-latency, serial, accesses to the page table. To this end, we extend GPUs with a hardware mechanism that memoizes virtual to physical translations directly in memory. Similar in design to the recently proposed PoM-TLB (Ryoo et al. 2017), we propose a TLB in GPU DRAM (DRAM-TLB) that serves as the next larger TLB in the processor TLB hierarchy whose contents are identical to the contents of on-chip TLBs. A DRAM-TLB is consulted upon LLT (or UCAT) misses before walking the page table.

Overall, this article makes the following contributions:

(1) We extend on-die TLB coverage by designing the conventional LLC to serve as a *Unified Cache and TLB (UCAT)*. UCAT holds cache lines and TLB entries in a single hardware structure. Thus, UCAT improves on-die TLB coverage by enabling as many on-die TLB entries as there are cache lines in the on-chip LLC.
(2) We reduce LLC miss penalty by architecting *TLBs in the GPU DRAM (DRAM-TLB)*. A DRAM-TLB is a very large hardware-managed structure that logically sits between the LLT (or UCAT) and the page table. A DRAM-TLB can be arbitrarily sized to provide the desired TLB coverage by only occupying a very small portion of gigantic DRAM sizes in today's systems. Unlike a page table walk, a DRAM-TLB provides low-latency address translation with a single memory access.
(3) We propose DUCATI, an address translation architecture for GPUs that combines DRAM-TLB and UCAT benefits to improve TLB coverage and LLT miss latency in high-performance computing systems.

For a set of high-performance computing workloads simulated on a heterogeneous CPU-GPU system with 4KB page size, Unified Cache and TLB (UCAT) improves performance by 65% on average (up to 4×). However, DRAM-TLB improves performance by 22% on average (up to 2.25×). Both proposals combined, DUCATI, improves performance by 81% (up to 4.5×). We also show that DUCATI requires negligible hardware changes and scales to larger page sizes and improves performance by 56% and 8% when using 64KB and 2MB pages, respectively.

2 BACKGROUND AND MOTIVATION

Virtual memory is ubiquitous in nearly all computing systems today and is responsible for translating virtual addresses to physical addresses. Address translation is performed by maintaining an application *page table* in system memory. Figure 2 shows a typical four-level hierarchical page table commonly used in computing systems today. Each node in the page table is 4KB in size and contains 512 eight-byte entries that point to the next node in the page table. The leaf node contains the physical address mapping. Generally, the page table is sparsely populated and new nodes are created only when data is referenced. As such, the typical in-memory storage space for the application page table is roughly 0.2% of the total memory footprint (8 bytes of storage overhead when using 4KB pages).

Figure 2 illustrates a *page walk* for a four-level page table. The Memory Management Unit (MMU) (or page walker) first consults a predetermined register (e.g., CR3 register on ×86 systems) to determine the physical address for Page Map Level 4 (PML4) or root node of the page table. The MMU indexes PML4 to fetch the Page Directory Pointer (PDP) or third-level page table (first

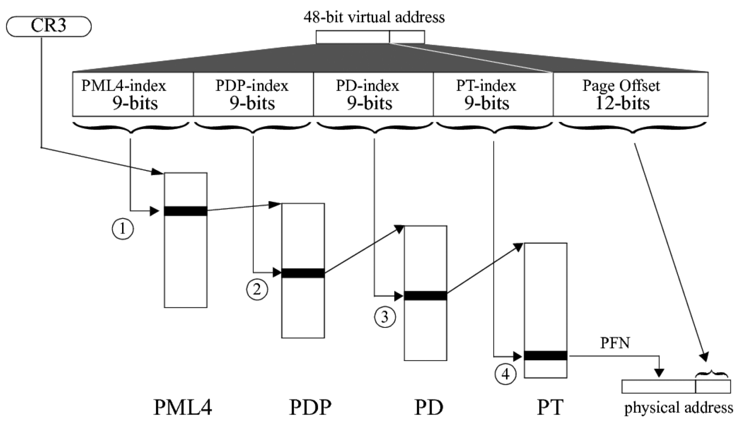


Fig. 2. A four-level hierarchical page table.

memory reference). Next, the MMU indexes the PDP to fetch the Page Directory (PD) or second-level page table (second memory reference). Next, the MMU indexes the PD to fetch the first-level of the Page Table (PT) (third memory reference). Finally, the MMU indexes the PT (fourth memory reference) to fetch the physical address mapped to the virtual address.

When walking the page table, commercial processors reduce memory accesses by using Page Walk Caches (PWC) for each level of the page table (Barr et al. 2010; Bhattacharjee 2013). PWCs exploit temporal and spatial locality in the page table access stream and avoid memory accesses on PWC hits. Consequently, the frequency of TLB misses and PWC hit rates impacts address translation performance. In the previous section, Figure 1 motivated increasing TLB coverage by demonstrating its direct impact on performance. To provide further insight, Table 2 shows the number of LLT misses for the evaluated workloads. We observe that these workloads incur frequent LLT misses when using 4KB pages. To make things worse, each LLT miss requires more than one memory access to the page table (maximum of two in some cases). Consequently, these workloads experience significant performance overheads due to TLB misses. With the number of levels in the page table hierarchy expected to grow to 5-levels (Corbet 2017), there is a need for reducing the number of memory accesses on an LLT miss in addition to improving LLT coverage. Before we describe our solutions to address both these problems, we first provide details on our experimental methodology.

3 EXPERIMENTAL METHODOLOGY

We assume a CPU-GPU heterogeneous system with a single CPU and a single GPU supporting shared virtual memory (Junkins 2015; Moammer 2016; GP100 2016). Figure 3 illustrates our baseline system where GPU LLT misses are handled by the CPU IOMMU using Address Translation Services (ATS) (Vesely et al. 2016; ATS 2009).

3.1 System Configuration

We use an industry proprietary performance simulator to simulate a GPU (Table 1) with a memory hierarchy similar to the NVIDIA Pascal GPU system (GP100 2016). We model 128 Streaming Multiprocessors (SM) that support 64 warps each. A warp scheduler selects warp instructions each cycle. The baseline GPU memory system consists of a multi-level non-inclusive cache and TLB hierarchy. The first-level cache and TLB are private to each SM while the memory-side LLC and LLT are shared by all the SMs (Bhattacharjee et al. 2011). All caches use 128B cache line size with 32B

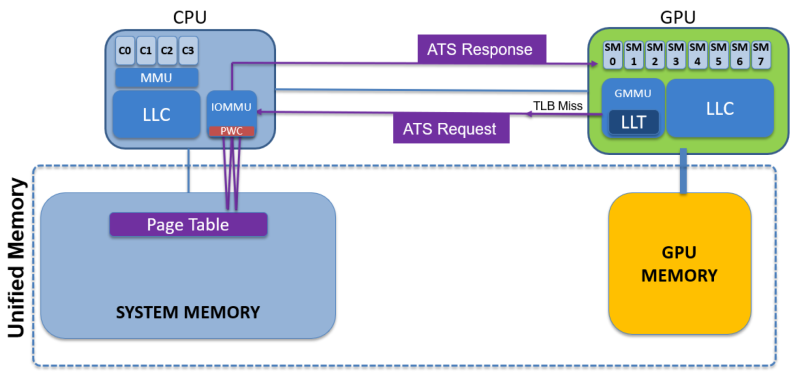


Fig. 3. Baseline system.

Table 1. Baseline System Configuration

Streaming Multiprocessor (SM)	
Frequency Number of SMs	1 GHz 128
Baseline TLB and Cache Hierarchy	
L1 TLB (private) LLT (shared)	32-entry, 4-way 1024-entry, 8-way
L1 cache (private) LLC (shared) MSHRs for Mem Reqs	128KB, 4-way 8MB, 16-way 128 per memory channel
Stacked Memory (HBM)	
Capacity Bus Frequency Channels / Banks Bus Width Row Buffer Size tCAS-tRCD-tRP-tRAS Total Bandwidth	16GB (4 stacks) 1GHz 32 / 16 per rank 128-bits per channel 2048 Bytes 14-14-14-33 1024GB/s
Main Memory (DDR4)	
Capacity Bus Frequency Channels / Banks Bus Width Row Buffer Size tCAS-tRCD-tRP-tRAS Total Bandwidth	256GB 1600MHz (DDR 3.2GHz) 8 / 16 per rank 64 bits per channel 2048 Bytes 14-14-14-35 204.8GB/s
Virtual Memory Configuration	
Page Allocation Page Table (PT) Page Walk Caches	Bandwidth-aware Placement 4-level hierarchical 16 entries per PT level

sectors. We assume 5 bytes of tag storage per LLC entry (including valid, dirty, coherence state, and replacement bits). For the UCAT architecture, we assume an extra 40 cycle load-to-use latency to consult the UCAT on an LLT miss. The baseline LLC (and UCAT) uses the DRRIP replacement policy (Jaleel et al. 2010) while the L1 cache and TLBs use pseudo-LRU replacement (Jaleel et al. 2010).

We model a hybrid memory subsystem consisting of 16GB of High Bandwidth Memory (HBM) technology (JEDEC 2013b) (referred to as stacked memory) and 256GB of CPU-attached commodity DRAM (referred to as system memory) using conventional DDR4 technology (JEDEC 2013a). We assume similar interconnect latency to the HBM controller and the DDR4 controller (50ns one-way). However, the stacked memory has 5× the bandwidth of system memory with similar random access latency. To ensure full utilization of the total available system bandwidth, physical pages are allocated based on the bandwidth ratio of the hybrid memory system (Agarwal et al. 2015; Chou et al. 2015a). In doing so, both system memory and stacked memory satisfy memory requests. The memory controller supports 128-entry read and write queues per channel, open-page policy, minimalist address mapping policy (Kaseridis et al. 2011) and FR-FCFS scheduling policy (prioritizing reads over writes). Writes are issued in batches when the write queue starts to fill up.

We model a virtual memory system that maps virtual addresses to physical addresses. We use a combination of bandwidth-aware page allocation (Agarwal et al. 2015; Chou et al. 2015a) and random page replacement policy. Our baseline system assumes 4KB page size; however, we also evaluate 64KB and 2MB page size. We model a four-level hierarchical page table (Barr et al. 2010): Page Map Level 4 (PML4), Page Directory Pointer (PDP), Page Directory (PD), and Page Table (PT). All application page tables reside in system memory where GPU LLT misses are serviced by the IOMMU on the CPU using ATS. We also evaluate a system where the page tables are stored in GPU memory and the GPU MMU (GMMU) services LLT misses instead. The IOMMU (and GMMU) is configured with a highly-threaded hardware page table walker (Power et al. 2014; Pichai et al. 2014) and on-chip Page Walk Caches (PWCs) (Barr et al. 2010; Bhattacharjee 2013) for each level of the page table. The PWCs are small, fully associative hardware structures that are indexed by portions of the virtual address (Bhattacharjee 2013). We model a 16-entry PML4-cache, 16-entry PDP-cache, a 16-entry PD-cache, and a 16-entry PT-cache (Bhattacharjee 2013).

3.2 Workloads and Metric of Interest

We use selected CUDA-based high-performance computing applications from CORAL (CORAL 2014), Mantevo (Heroux et al. 2009), and LoneStar (Kulkarni et al. 2009) suites (see Table 2). Our study also includes a graph-traversal workload (*MaxFlow* (Jiang et al. 2013)) and a random memory access workload (*GUPS* (Luszczek et al. 2006)). All workloads are run with large inputs to stress the hybrid memory system. We collected representative program regions and warm up the caches, TLBs, and PWCs by executing four billion warp instructions. After functional warmup, we enable detailed timing simulation for two billion warp instructions.

Reduction in total execution time is our primary metric for performance. We also compare address translation latency to correlate the change in performance. We compute translation latency as the cycles spent in translating a virtual address to physical address averaged across all memory instructions executed by the application.

4 UNIFYING CACHES AND TLBS

Modern chip multiprocessors use multi-level TLB and cache hierarchies for high performance. The last-level in each hierarchy is architected as a single large unified structure that holds both instruction and data entries and is shared among all cores. For example, the unified LLC contains hundreds of thousands of cache lines (e.g., 256k lines in an 8MB cache with 32B cache lines). Similarly, the unified LLT contains 512–1024 TLB entries (Sodani et al. 2016; Intel 2009).

Table 2. TLB Behavior 4KB Page Size on Simulated 128 SM GPU

LLT Sensitivity	Name	LLT-mpki	PT Access/LLT Miss	Total Footprint
High	XSBench	90.9	1.59	3.2GB
	dmr	81.3	1.81	1.5GB
	GUPS	92.0	2.00	15GB
	MaxFlow	86.8	1.83	1.5GB
	MCB	69.5	1.00	100MB
Medium	bfs	1.45	1.06	572MB
	LULESH	4.64	1.49	3.7GB
	SNAP	4.81	1.15	1.6GB
	UMT	3.71	1.51	1.4GB
	CoMD	4.95	1.18	780MB
	HPGMG	6.78	1.54	4.4GB
Low	MiniAMR	5.58	1.62	3.3GB
	AMG	2.30	1.38	2.9GB
	Nekbone	4.36	1.29	1.1GB

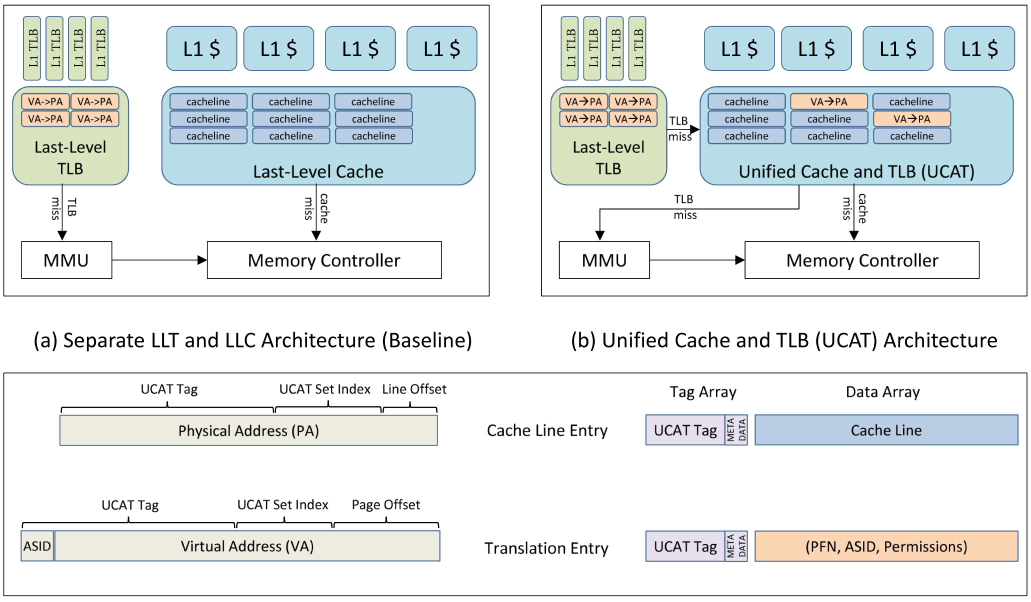
The LLT and LLC are cache structures with a tag array and a data array. The LLT tag array stores virtual addresses while the LLC tag array stores physical addresses.[2] The LLT data array stores the physical address corresponding to the virtual address translation (roughly 8 bytes), while the LLC data array stores a copy of the data from memory at a cache line granularity (e.g., 128-byte lines with 32 byte sectors).

The LLT and LLC have similar sized tag arrays. However, since they provide different types of data, the LLC has a 4×–8× larger data array. To increase on-die LLT capacity, we observe that the LLC can potentially also serve as gigantic TLB with as many entries as there are LLC lines. Given that LLCs typically have low hit-rates and are often inefficiently used[3] (Lee and Kim 2012; Jaleel et al. 2010; Qureshi et al. 2007; Wu et al. 2011; Jiménez 2013; Khan et al. 2010), the conventional LLC can be used to store virtual to physical translations just like a TLB (in addition to caching data from memory). Based on this insight, we propose to re-architect the conventional LLC as a *Unified Cache and TLB (UCAT).*

4.1 UCAT Architecture

Figure 4(a) illustrates a baseline system with separate LLT and LLC structures, while Figure 4(b) demonstrates a system with a separate LLT and a Unified Cache and TLB (UCAT). In the latter system, the UCAT holds both cache lines and TLB entries in a single hardware structure. Figure 4(c) shows how this is done. When a UCAT entry stores a cache line similar to what the baseline system does, the tag-array stores a portion of the physical address while the data array stores the cache line (e.g., 32-bytes).[4] However, when the UCAT entry serves as a TLB entry, the UCAT tag-array stores portions of the virtual address and the address space identifier (ASID), while the data-array stores the virtual to physical address translation, and additional information such as page protection bits, and possibly some ASID bits. Storing some ASID bits in the data array may be necessary if all the ASID bits do not fit in the existing LLC tag array size. In such situations, we propose to store the

[2]Additional meta data (e.g., coherence state, replacement state, etc.) is also stored in the tag array.
[3]For the workloads in this study, on average 80% of the lines installed in the LLC are never re-referenced.
[4]In our baseline we assume a physically indexed, physically tagged cache. However, any other variant of virtual or physical indexing/tagging is equally possible with UCAT.



(c) UCAT is Indexed using Physical Address to Retrieve Data and using Virtual Address to Retrieve Translations. The contents of the UCAT Tag and Data Array are also illustrated.

Fig. 4. UCAT architecture.

top bits of the ASID in the tag array. Doing so enables a partial tag match and avoids false positives when multiple programs with identical virtual addresses concurrently execute on the system. Note that the bottom bits of the ASID (stored in the data array) must be compared for a match before supplying the translation stored in the UCAT-entry. We use 16 bytes of the UCAT data array for storing the address translation.

Even though we only use 16-bytes of the 32-byte sector to store the translation, UCAT improves upon the existing LLC inefficiency. This is because recent CPU and GPU studies show (and this study independently verifies) that the majority of LLC entries are unused after cache insertion (Lee and Kim 2012; Jaleel et al. 2010; Qureshi et al. 2007; Wu et al. 2011; Jiménez 2013; Khan et al. 2010). As such, UCAT utilizes the conventional LLC space more efficiently by storing TLB entries in addition to cache lines. Note that UCAT space efficiency can be improved further by compressing multiple TLB entries in the unused portion of the data array. However, we leave these optimizations for future work.

We now discuss the lookup and fill operations for a UCAT. When the GPU misses in the LLT, it consults the UCAT to determine if the missing translation is in the UCAT. If so, then the translation is returned to the LLT. However, on a UCAT miss, the request is sent to the MMU to walk the page table and retrieve the missing translation. The missing translation is then inserted into the UCAT (for future hits) and also returned to the LLT. For the purpose of our evaluations we assume a non-inclusive TLB hierarchy.

In a UCAT, cache lines and TLB-entries dynamically contend for the available UCAT space. Like the baseline LLC architecture, we leverage the existing replacement policy to manage allocating UCAT entries in a set. UCAT hits update replacement state while UCAT misses utilize the baseline replacement policy to select the victim. The baseline UCAT replacement policy allocates entries based on demand, rather than utility. This can potentially create performance problems in

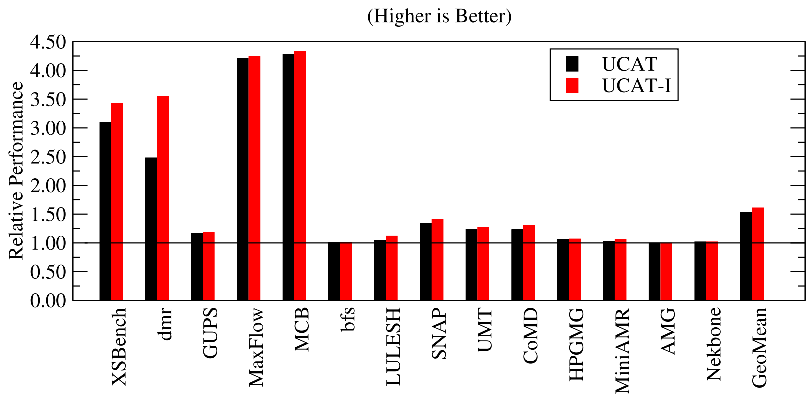


Fig. 5. UCAT performance.

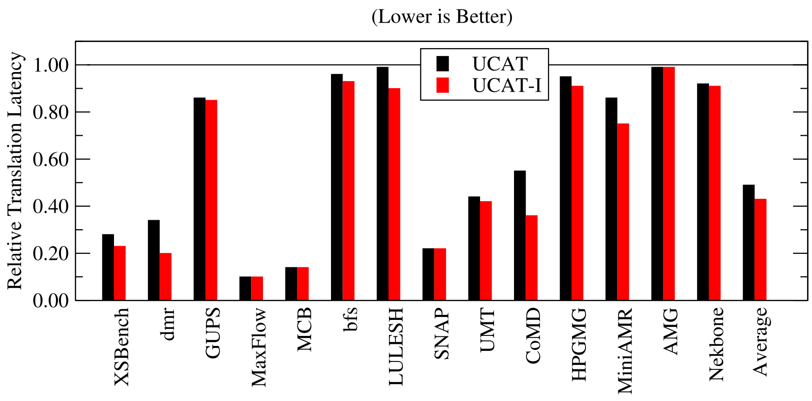


Fig. 6. Relative translation latency.

TLB-sensitive workloads that frequently stream through a large number of cache lines and constantly discard performance critical UCAT TLB entries. Based on this insight, we enhance the baseline UCAT design by improving the replacement policy. To this end, we leverage the Dynamic Re-Reference Interval Prediction (DRRIP) replacement policy (Jaleel et al. 2010). In this policy, all UCAT insertions follow the same insertion policy. We propose enhancing the insertion policy for TLB entries by inserting them with a *near-immediate prediction* rather than the default *far prediction*. By doing so, TLB entries reside in the UCAT for a longer duration. We refer to this enhancement as *UCAT with Insertion (UCAT-I)*.

4.2 UCAT Performance

Figure 5 shows UCAT performance relative to the baseline system(on the y-axis) with workloads on the x-axis. The first bar in the figure shows UCAT performance where cache lines and TLB-entries contend for UCAT space without any restrictions. The figure shows that UCAT significantly improves performance of workloads like *XSBench*, *dmr*, *MaxFlow*, and *MCB* by more than 2× (up to 4× in the latter two). Other workloads like *GUPS*, *SNAP*, *UMT*, and *CoMD* observe more than 15% performance gains. Overall, UCAT improves performance across all workloads by 53%.

To understand the performance benefits of UCAT, Figure 6 and Figure 7 illustrate relative translation latency and effective on-die TLB size respectively. The effective on-die TLB size is reported

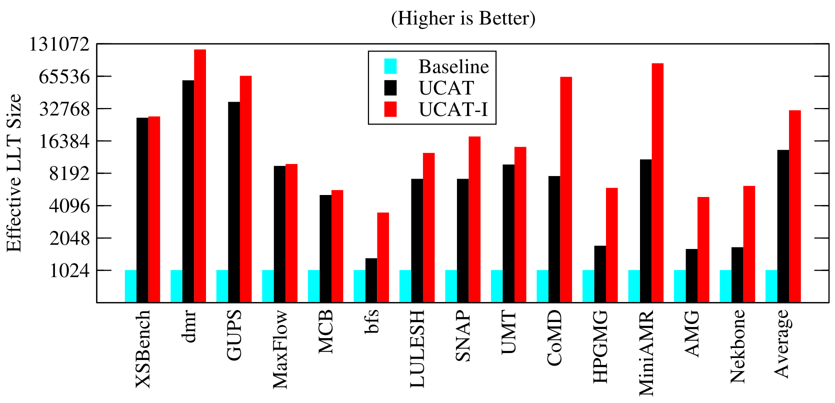


Fig. 7. Effective on-die TLB size with UCAT.

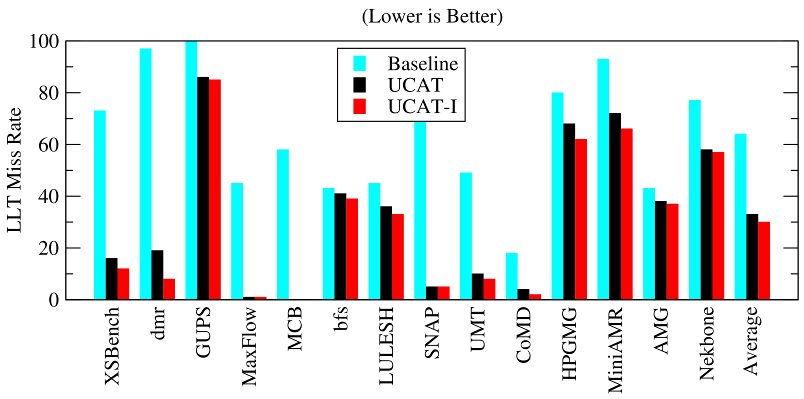


Fig. 8. On-die TLB miss rate.

as the average number of TLB entries in the UCAT (sampled every 10 million cycles) during the course of application execution.

Workloads with more than 2× performance improvement observe a reduction in TLB miss penalty by 70% or more. This is because these workloads experience a 4×-64× increase in on-die TLB size. On average, UCAT improves address translation latency by 51% (up to 86% for *MCB*). The improvement in translation latency stems from an increase in on-chip TLB coverage by 16× (up to 64× for *dmr*). We note that some workloads (e.g., SNAP) experience modest performance benefits despite a significant reduction in TLB miss penalty. This is because SNAP has abundant parallel work that can be performed in the shadow of a TLB miss. As such, SNAP despite its frequent TLB misses is fairly insensitive to TLB miss latency.

Figure 5 also shows that intelligently managing UCAT by prioritizing TLB-entries over cache lines improves average UCAT performance by an additional 12%. We observed that UCAT-I also has negligible increase in memory traffic (<1%) and TLB-sensitive workloads like *XSBench* and *dmr* experience additional performance improvements of up to 90%. Additionally, workloads like *LULESH*, *SNAP*, and *CoMD* experience an additional 7–10% performance improvement over UCAT by intelligently managing the UCAT-entries. This is evident from Figures 6 and 7 where UCAT-I increases effective TLB coverage by nearly 2× while improving average translation latency by 7%. Figure 8 shows that the increase in effective TLB size halves the baseline TLB miss rate from

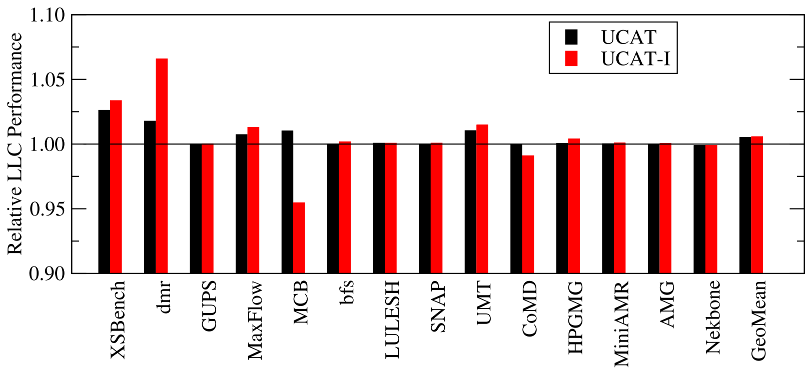


Fig. 9. The impact of UCAT on LLC performance.

roughly 60% to about 30%. Overall, the reduction in misses improves UCAT-I performance by 65% relative to the baseline.

4.3 UCAT Impact on LLC Performance

Since UCAT decreases the effective capacity of cache lines compared to the separate LLC, we investigated the increase in memory traffic relative to the baseline design. Figure 9 show that on average, UCAT increases the number of memory accesses by 0.5% (maximum 5% for *dmr*). We find the additional increase in memory traffic has negligible performance impact. This illustrates that trading off conventional LLC space for TLB entries provides substantial performance improvements.

4.4 UCAT Design Overhead

UCAT requires a mechanism to distinguish between cacheline entries and translation entries. We propose using the UCAT-entry state bits (i.e., MESI bits) and insert TLB-entries into UCAT with the *exclusive* and *shared* status bits both set to valid. Since cache lines can either be in exclusive state or shared state and not both, this proposal enables distinguishing between TLB entries and cacheline entries at zero storage overhead. Besides distinguishing between cacheline and translation entries, UCAT also requires efficient support for handling TLB shootdown and TLB flush requests. We discuss these design issues in Section 7.

5 ARCHITECTING TLBS IN DRAM

When the application working set exceeds the TLB coverage provided by UCAT (e.g., in *GUPS*), UCAT does not reduce the number of memory accesses on an LLT miss. Reducing the number of memory accesses on an LLT miss reduces effective LLT miss latency. As such, we now investigate increasing TLB coverage by extending the TLB hierarchy using *TLBs in DRAM (DRAM-TLB)*. DRAM-TLB logically sits between the on-die shared LLT (or UCAT) and the page tables in memory. DRAM-TLB is first consulted on an LLT (or UCAT) miss before walking the page table (see Figure 10).

5.1 DRAM-TLB Architecture

DRAM-TLBs can either be architected using stacked memory or system memory. A DRAM-TLB architected using stacked memory is referred to as a *Stacked-TLB* while a DRAM-TLB architected using system memory is referred to as a *SysMem-TLB* (Ryoo et al. 2017). DRAM-TLBs are physically

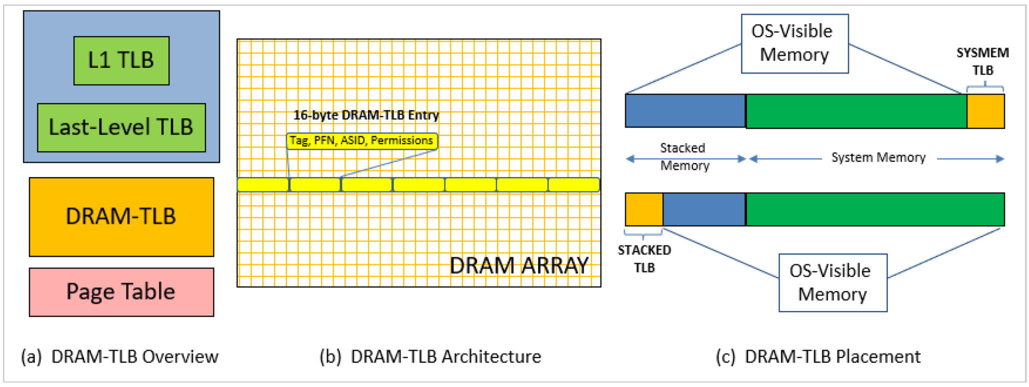


Fig. 10. Improving TLB coverage by embedding TLBs in DRAM (DRAM-TLB). A DRAM-TLB architected using commodity DRAM is called SysMem-TLB and a DRAM-TLB architected with stacked DRAM is called Stacked-TLB.

placed in a large contiguous segment of memory and are entirely hardware-managed. As such, DRAM-TLBs do not contribute to the OS-visible memory space (see Figure 10(c)).

5.1.1 DRAM-TLB Organization. Like conventional on-chip SRAM TLBs, and the proposed UCAT architecture, a DRAM-TLB entry maintains the TLB tag, physical address, and meta data information such as valid bits, page permission bits, address space identifier (ASID). To accommodate this information, we use 16 bytes of storage per DRAM-TLB entry. Figure 10 illustrates the layout of the DRAM-TLB in the DRAM array. For example, a 2KB DRAM row buffer holds 128 DRAM-TLB entries per row.

5.1.2 DRAM-TLB Lookups. Unlike on-chip TLBs, all DRAM-TLB operations occur on the DRAM data bus at the granularity of the DRAM interface. Thus, on an LLT miss, if implemented as a stacked-TLB, the lookup fetches data at the granularity of the stacked memory interface: 32-byte cacheline (two TLB entries), and if implemented in system memory, the lookup fetches data at the granularity of the DDR interface: 64-byte cacheline (four TLB entries).

Reading multiple TLB entries enables architecting a set-associative DRAM-TLB without incurring additional latency or bandwidth (Qureshi and Loh 2012; Loh and Hill 2011). We evaluate both set-associative and direct-mapped designs and observe similar performance among these design points with large DRAM-TLB sizes. These results match the behavior of existing work on large DRAM Caches where conflict misses tend to be low (Qureshi and Loh 2012). Thus, we architect a direct-mapped DRAM-TLB where consecutive DRAM-TLB sets map to the same row buffer in memory (to exploit DRAM row buffer locality). Adjacent DRAM-TLB entries fetched are also inserted into the PWC for potential future hits.

We could further take advantage of reading multiple TLB entries with a single read operation. Retrieving the co-located DRAM-TLB entries on a DRAM-TLB read naturally enables prefetching of neighboring address translations. For example, prefetched entries can be stored in an on-chip TLB prefetch buffer. This approach is similar to caching co-located page table entries in the PWCs on a conventional page table walk. We leave the evaluation of such designs to future work.

Designing DRAM-TLB as a direct-mapped structure can incur frequent conflict misses especially when multiple virtual addresses (from the same or different processes) map to the same DRAM-TLB set. Further enhancements such as hashing the lower order bits of the set index with the ASID and coordinating way installation and way prediction (Young et al. 2018a) can address conflict misses

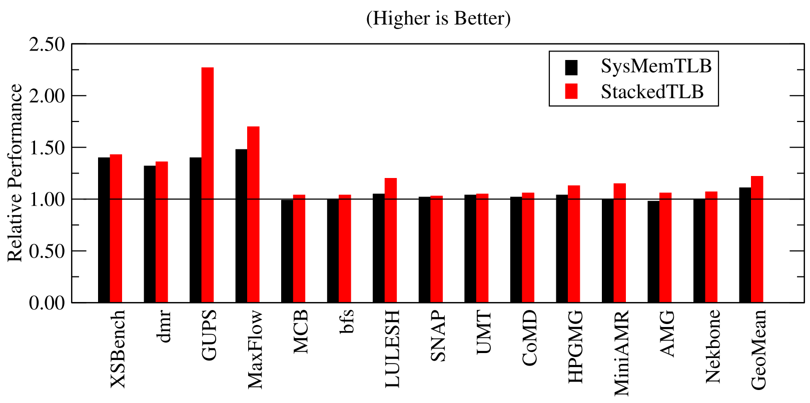


Fig. 11. Performance of DRAM-TLBs.

in direct-mapped DRAM-TLBs. In practice, we find conflict misses are rare (see Figure 14). As such, evaluation of such designs are also left for future work.

5.1.3 DRAM-TLB Insertions and Updates. The DRAM-TLB must be updated on misses, shootdowns, and page permission changes. We propose to modify the TLB-entry directly in DRAM on updates. To ensure that updating a single DRAM-TLB-entry does not corrupt the contents of co-located DRAM-TLB entries, we leverage existing DRAM interfaces that allow partial writes (e.g., byte-level writes) to memory without the power and bandwidth overhead of a full read-modify-write operation (JEDEC 2013b).

5.1.4 DRAM-TLB Implementation. Since the DRAM-TLB is stored in physical memory, the physical address for the DRAM-TLB entry must be computed. We use an 8-byte register in the Memory Management Unit (MMU) to store the base address (*BaseAddr*) for the DRAM-TLB. Given the DRAM-TLB set index *SI*, DRAM-TLB associativity *A*, and DRAM-TLB entry size *s* (16 bytes in our case), the MMU computes the physical address for the DRAM-TLB entry using combinational logic:

$$\text{PhysAddr} = BaseAddr + SI * A * s. \tag{1}$$

Once the DRAM-TLB entry is retrieved from memory, the MMU compares the tag to determine hit or miss. On a DRAM-TLB hit, the missing translation is returned to the processor. However, on a DRAM-TLB miss, the MMU walks the application page table to determine the virtual to physical translation. This translation is returned to the processor and also inserted into the DRAM-TLB.

5.1.5 DRAM-TLB Example. Assume a 1M-entry direct-mapped DRAM-TLB (with 4KB pages) starting at memory location *BaseAddr=0*. An LLT miss for virtual page 0xff2212345000 requires fetching the 16-byte DRAM-TLB entry at set index 0x12345. With $A = 1$ and $s = 16$, the physical memory location for this DRAM-TLB entry is 0x123450 (i.e., 0 + 0x12345 * 1 * 16). The MMU compares the missing tag (0xff22) with the DRAM-TLB entry to determine DRAM-TLB hit or miss.

5.2 DRAM-TLB Performance

We evaluate SysMem-TLB and Stacked-TLB performance assuming an 8-million entry DRAM TLB (32GB TLB coverage with 4KB pages). Figure 11 shows that DRAM-TLBs improve performance relative to the baseline system across all TLB-sensitive workloads. On average, SysMem-TLB improves performance by 11% while Stacked-TLB improves performance by 22%.

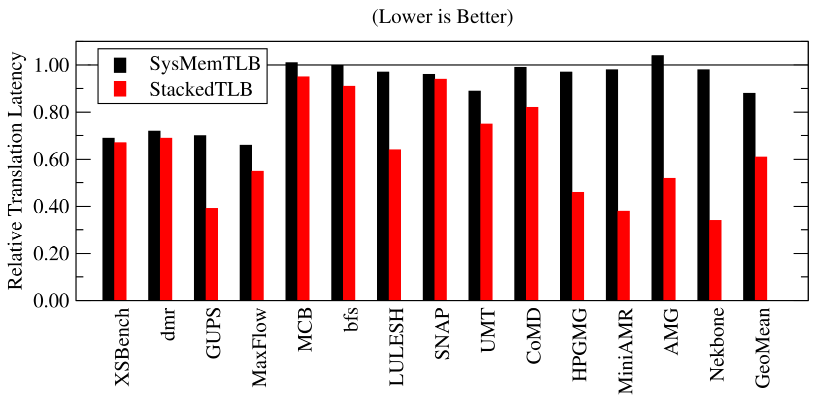


Fig. 12. Relative translation latency.

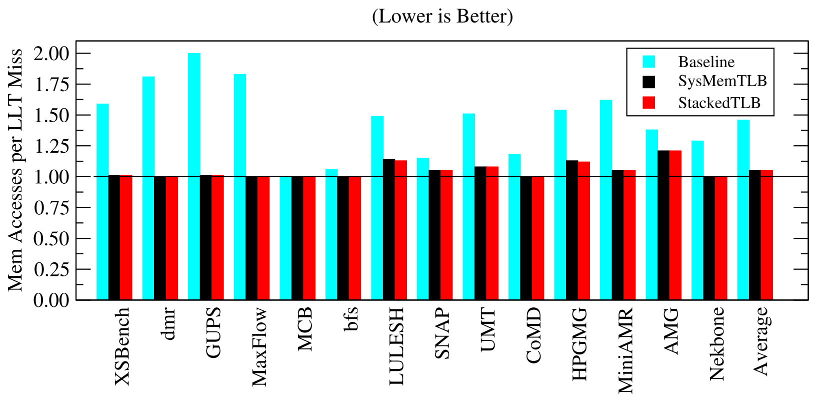


Fig. 13. Memory accesses on an LLT miss.

Figure 12 shows that SysMem-TLBs reduce address translation latency by 12% while Stacked-TLBs reduce address translation latency by 40%. In general, Stacked-TLBs perform better than SysMem-TLBs, because they are implemented with stacked memory, a technology that provides lower memory queuing delay due to the high memory bandwidth. In fact, our evaluations show that the loaded stacked memory latency tends to be an order of magnitude lower than system memory access latency.

DRAM-TLBs decrease address translation latency by reducing the number of memory accesses on an LLT miss. Figure 13 shows that the baseline system incurs 1.5 memory accesses per LLT miss (up to 2) on average. However, both SysMem-TLB and Stacked-TLB incur only 1.05 memory access per LLT miss (max of 1.21) on average. These results correspond to a near 100% hit rate in the DRAM-TLBs (see Figure 14), implying that the only memory access required on an LLT miss is to fetch the translation from the DRAM-TLB. Note that minor differences in memory accesses per LLT miss between SysMemTLB and StackedTLB are due to timing variations that cause conflict misses in page walk caches. Overall, DRAM-TLBs boost performance of TLB-sensitive workloads like *GUPS* by 2.2× and *MaxFlow* by 70%. Furthermore, workloads like *XSBench*, *dmr*, *LULESH*, and *MiniAMR* experience more than 15% performance gain. These results show that DRAM-TLBs decrease LLT miss latency by reducing the number of memory accesses on an LLT miss.

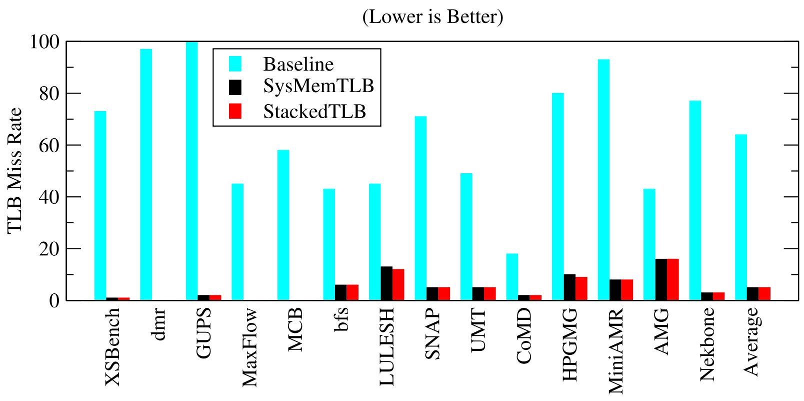


Fig. 14. TLB miss rate for the last level TLB in the TLB hierarchy.

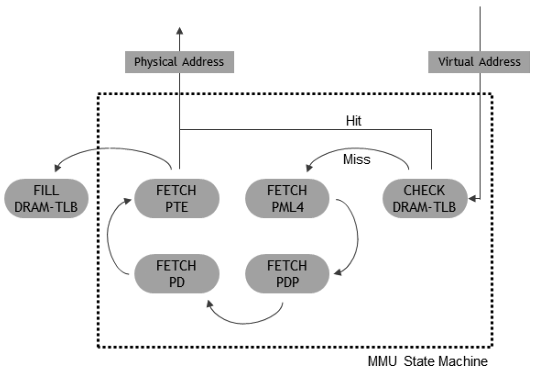


Fig. 15. GMMU extensions to support DRAM-TLBs.

5.3 DRAM-TLB Design Overhead

Figure 15 shows simple modifications to the GPU Memory Management Unit (MMU) state machine to support DRAM-TLBs. On an LLT miss, the GMMU first consults the DRAM-TLB to determine if the translation is already present in the DRAM-TLB. If the request misses in the DRAM-TLB, then the MMU walks the different levels of the page table. Thus, we simply introduce a new state in the MMU state machine for the DRAM-TLB lookup before walking the page table.

The in-memory storage overhead for DRAM-TLBs depends on the desired TLB coverage. For example, achieving full system memory (256GB in our baseline) coverage with 4KB, 64KB, and 2MB pages requires 1GB, 64MB, and 2MB of storage overhead, respectively (assuming 16-byte DRAM-TLB entries). In general, this storage overhead is impractical for on-chip SRAM TLBs. However, these sizes are an insignificant fraction of emerging multi-gigabyte DRAM systems. For example, the aforementioned storage overheads correspond to 6% (4KB pages), 0.4% (64KB pages), and 0.01% (2MB pages) storage overhead for a 16GB stacked memory system. Consequently, DRAM-TLBs improve TLB coverage using small pages with minimal storage overhead and require no significant changes to the existing address translation architecture.

Like UCAT, DRAM-TLBs also require efficient support for handling TLB shootdown and TLB flush requests. We discuss these design issues in Section 7.

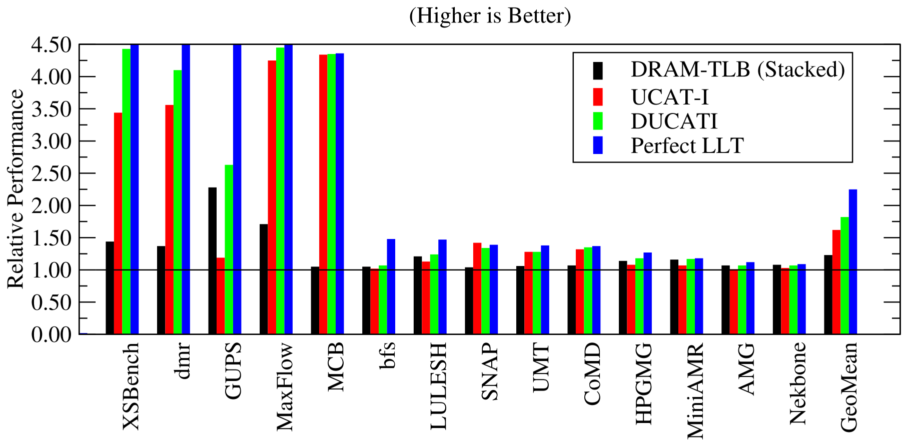


Fig. 16. Performance summary (4KB page size).

6 COMBINING DRAM-TLBS AND UCAT

UCAT and DRAM-TLB independently improve on-die processor TLB coverage and LLT miss overhead, respectively. These mechanisms can be combined to collectively improve processor performance. To this end, we propose *DUCATI*, an address translation architecture that combines DRAM-TLBs and UCAT-I. While DUCATI can be architected with Stacked-TLBs or with SysMem-TLBs, we limit our analysis of DUCATI to Stacked-TLBs. This is because stacked memory technology provides better bandwidth and latency than conventional DDR memory technology.

6.1 DUCATI Performance

Figure 16 illustrates the relative performance of our proposals to the baseline system. The x-axis shows the different workloads while the y-axis illustrates performance. For each workload, we present the relative performance for DRAM-TLBs architected with stacked memory (i.e., Stacked-TLB), Unified Cache and TLB (UCAT) enhanced with insertion (UCAT-I), DUCATI, and for comparison, a hypothetical Perfect LLT system.[5] To provide insight into the reason for performance improvement, Figure 17 presents relative address translation latencies of the different proposals.

Figure 16 shows that stacked memory DRAM-TLBs improve average performance by 22%, UCAT with insertion (UCAT-I) improves average performance by 61%, while DUCATI combines the benefits of both to improve average performance by 81%. In fact, workloads like *XSBench*, *dmr*, *MaxFlow*, and *MCB* experience 4× or more performance improvement. Most of the performance gain for these workloads is due to increase in on-die LLT coverage. However, *GUPS* sees a 2.5× performance boost due to reduced LLT miss penalty (i.e., the DRAM-TLB component of DUCATI).

Figure 17 illustrates the relative address translation latency for the different proposals. On average, DRAM-TLB and UCAT reduce address translation latency by 40% and 60%, respectively, while DUCATI reduces address translation latency by 75%. In doing so, DUCATI improves performance by 1.81× while a perfect LLT system improves performance by 2.24×. Thus, DUCATI bridges two-thirds of the performance gap between the baseline system and a perfect LLT system.

[5]We model a Perfect LLT by assuming all references to the LLT in the baseline system are hits.

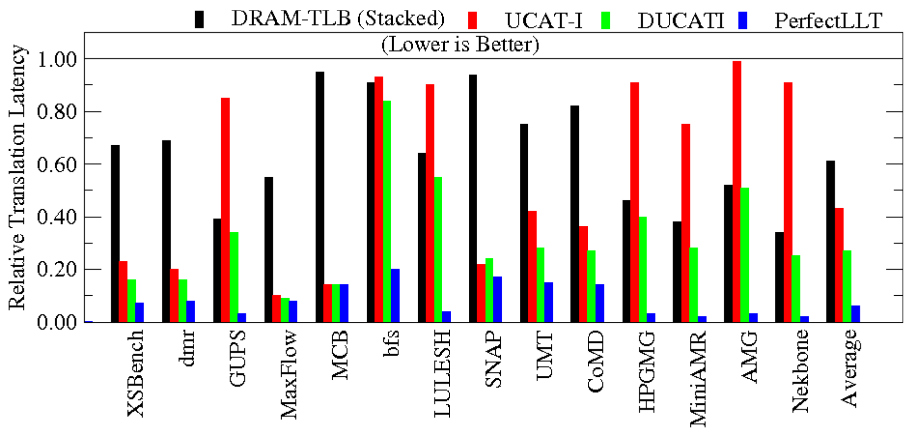


Fig. 17. Translation latency (4KB page size).

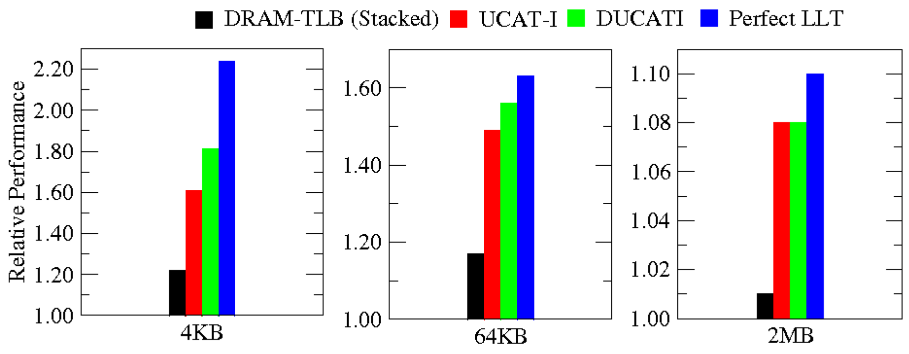


Fig. 18. Performance sensitivity to page size.

6.2 DUCATI Sensitivity to Page Size

We now discuss the performance of our proposals using larger page sizes. Note that while large page sizes improve TLB coverage, unrestricted use of them can create various performance overheads (Talluri and Hill 1994; Talluri et al. 1992; Gaud et al. 2014; Chou et al. 2014; Pham et al. 2015). Nonetheless, Figure 18 illustrates the average performance across all workloads for 4KB, 64KB, and 2MB pages. Overall, DUCATI improves performance by 81% with 4KB pages, 56% with 64K pages, and 8% with 2MB pages. In fact, DUCATI is within 20%, 5%, and 2% the performance of an unrealistic perfect LLT system when using 4KB, 64KB, and 2MB pages, respectively. Thus, DUCATI bridges the performance gap between the baseline system and a perfect LLT system regardless of page size.

6.3 DUCATI Sensitivity to LLC Size

Table 3 presents DUCATI performance averaged across all our workloads with 4KB pages and LLC sizes ranging from 2MB to 32MB. The table shows that DUCATI improves performance across all LLC sizes studied. In general, performance increases with growing LLC size. For example, performance improves by 61% with a 2MB LLC while performance improves by 93% with a 32MB LLC. This is because larger LLCs enable more translation entries to be cached in the UCAT structure. Overall, we observe that DUCATI improves performance regardless of the system LLC size.

Table 3. DUCATI Sensitivity to LLC Size (4KB Pages)

LLC Size	2 MB	4 MB	8 MB	16 MB	32 MB
Speedup	1.61	1.73	1.81	1.89	1.93

Table 4. Performance when GPU Walks Page Table

Page Size	UCAT	DRAM-TLB	DUCATI
4KB	1.20	1.16	1.32
64KB	1.26	1.20	1.33
2MB	1.01	1.00	1.01

7 DISCUSSION

Thus far we have shown that DUCATI can improve TLB miss overhead when the GPU relies on the CPU for addresses translation support. We now investigate DUCATI performance when the GPU itself can walk the page table. We also discuss how DUCATI can potentially impact the behavior of virtual memory operations such as TLB shootdowns and TLB flushes.

7.1 Performance When GPU Walks Page Table

We also investigate the performance of our mechanisms when the application page table is placed in GPU memory. In this scenario, the GPU MMU (and not the CPU IOMMU) walks the application page table. In doing so, the GPU MMU now has high-bandwidth and low-latency access to the application page table. Furthermore, a GPU page table walk has additional locality benefits of caching the application page table in the memory-side GPU LLC. Table 4 presents the average performance of UCAT, DRAM-TLB, and DUCATI for the different page sizes when the GPU walks the application page table on an LLT miss. The table shows that DUCATI improves performance by 32%, 33%, and 1% for the 4KB, 64KB, and 2MB pages, respectively. These results show that there is still significant opportunity to improve TLB miss overheads even when the GPU walks the application page table.

7.2 Impact on TLB Shootdowns

Virtual memory operations such as remapping the virtual to physical address or changes to page permissions require efficient handling of page table update requests such as TLB shootdowns. It would be highly desirable that DUCATI does not impact the overall latency of TLB shootdowns.

TLB shootdowns normally occur as a part of an interrupt flow that requires 5–30μs (Zheng et al. 2016). The access latency for UCAT and DRAM-TLB tends to be a few ten to a few hundred nanoseconds. Thus, even in the worst case, the additional shootdown latency is a small fraction of the overall interrupt latency (2–5%). As such, DUCATI has negligible performance impact on TLB shootdown latency.

7.3 Impact on TLB Flushes

TLB flushes normally occur when the virtual to physical mappings of an entire ASID need to be updated. Flushing conventional on-die TLB sizes (e.g., 1,024 entries in the baseline) can be done quickly. However, flushing thousands of TLB entries (as in UCAT) or millions of TLB entries (as in DRAM-TLB) can be a very long latency operation. To address this problem, we provide architecture support to flush large TLBs with zero latency overhead. To this end, we propose to associate an *Epoch Counter (EPCTR)* with each ASID. This 4-byte counter tracks the virtual to physical translations for an ASID in a given epoch. In the DUCATI framework, the DRAM-TLB and UCAT store the EPCTR (in addition to the physical address and page permission bits) with each TLB entry. When the ASID requires a TLB flush, the EPCTR can simply be incremented. Thus, DUCATI provides a valid translation only if the ASID and EPCTR stored in the TLB-entry match the current EPCTR of the ASID. Both DRAM-TLB and UCAT have sufficient storage space to support a 16-bit EPCTR per entry. As such, the TLB flush overhead can be completely eliminated with negligible hardware (4 bytes per ASID) and zero latency overhead.

7.4 Handling Multiple Page Sizes

On computing systems that simultaneously support multiple page sizes, DUCATI requires support for translating multiple page sizes. A naive implementation would require consulting the UCAT and DRAM-TLB for each page size for hit/miss confirmation. To avoid the bandwidth and latency for the multiple accesses, we leverage recent work that uses a simple predictor to learn the page size for any given virtual address (Ryoo et al. 2017). In this work, the authors propose a 512-entry table of 2-bit saturating counters indexed by the lower bits of the virtual address. The value of the 2-bit counter predicts the page size: small or large. Mispredicts still incur the latency and bandwidth of multiple memory accesses, however in the common case we find that such a predictor can provide near-perfect predictions on GPUs in steady state.

8 RELATED WORK

While significant literature exists on improving TLB performance, we discuss recent work that is most closely related to the work described in this article.

Improving TLB Performance. Recent work improve TLB coverage through compression (Pham et al. 2012, 2014) and enhanced TLB organizations (Bhattacharjee et al. 2011; Chen et al. 1992) that modify the existing TLB structures. Other work has investigated mechanisms to accelerate page walks by caching portions of the page table (Barr et al. 2010; Bhattacharjee 2013), prefetching TLB entries (Bhattacharjee and Martonosi 2010; Kandiraju and Sivasubramaniam 2002; Saulsbury et al. 2000; Power et al. 2014), or speculating on the address translation on TLB misses (Barr et al. 2011). When the memory footprint of workloads is extremely large, such proposals are unable to avoid TLB miss overhead. Our work focuses both on increasing TLB coverage and reducing TLB miss overheads by storing TLB entries in the conventional LLC (i.e., UCAT) and embedding a TLB in DRAM (i.e., DRAM-TLB). UCAT is similar to prior work on in-cache address translation (Wood et al. 1986) where a VIVT L1 cache is used to store the PTE entry. This work is only applicable to VIVT caches, requires page tables to be aligned at a fixed granularity (e.g., 256MB), and is primarily targeted on reducing TLB hardware without sacrificing TLB coverage. However, UCAT significantly improves TLB coverage and can be applied to any level of the cache hierarchy (VIVT or PIPT) and requires no restrictions on page table placement.

DUCATI has two components: UCAT and DRAM-TLB. DRAM-TLBs resemble the software-managed SPARC Translation Storage Buffer (TSB) architecture (Feehrer et al. 2013). Unlike the SPARC TSB, DRAM-TLBs are entirely hardware-managed and incur no software overhead (e.g., expensive traps). DRAM-TLBs also resemble recent CPU-centric work that extends TLB reach of CPUs (referred to as PoM-TLB) (Ryoo et al. 2017). Our work explores the DRAM-TLB design space on GPUs when architected in CPU memory space as proposed in the original work. However, we show that architecting TLBS in CPU memory (see Section 5.2) incurs additional latency and bandwidth overhead for accessing CPU memory. This is due to the low bandwidth system level interconnection network to access system memory from the GPU. As such, we propose architecting TLBs in the GPU attached high bandwidth memory. Unlike PoM-TLB, we also extend DRAM-TLBs with support for virtualized environments. Specifically, we propose low overhead TLB flushes without incurring any additional performance overheads on context switches.

PoM-TLB also improves on-die translation coverage by caching portions of the PoM-TLB in the on-chip last level cache. However, this proposal requires changes to the existing memory controller to physically support a PoM-TLB. Our UCAT work however is independent of a large memory-resident TLB. Instead, UCAT requires localized changes to the existing on-chip LLC that enable the LLC to serve simultaneously as a data cache *and* as a TLB. While on-chip caching of PoM-TLB and UCAT share the same spirits, the architecture and implementation of UCAT is entirely different from caching portions of the PoM-TLB. UCAT is a novel TLB and LLC structure for improving TLB reach while PoM relies on caching recent accesses in the LLC. We show that the combination of DRAM-TLBs and UCAT are complementary and additive. As such, we believe that UCAT can provide additional improvements with PoM-TLB when applied to CPU systems.

Large Pages and Direct Segments. Our studies show that the use of large pages (e.g., 2MB, 64MB, 1GB) and direct segments (Basu et al. 2013) can significantly improve TLB coverage. However, recent work has shown that unrestricted use of large pages creates unintended performance overheads (Talluri and Hill 1994; Talluri et al. 1992; Gaud et al. 2014; Chou et al. 2014; Pham et al. 2015) in emerging NUMA GPU systems (Young et al. 2018b). Alternatively, the use of direct segments can improve TLB coverage. However, direct segments requires both hardware and software support to redesign the existing address translation system. Our proposals provide a complementary approach to improve address translation using smaller pages (4KB, 64KB) without the overhead of large pages and the design complexity of direct segments.

Alternate DRAM Architectures. Recent proposals extend the processor cache hierarchy by architecting stacked memory as a DRAM cache (Chou et al. 2015b; Qureshi and Loh 2012; Jevdjic et al. 2014; Loh and Hill 2011; Sim et al. 2012, 2013). The tradeoffs for architecting DRAM-TLBs are different from those considered in DRAM cache designs. For example, DRAM-TLB entries are much smaller than DRAM cache entries and as such, DRAM-TLBs expose different tradeoffs for reading and updating multiple entries without incurring higher latency or bandwidth. Overall, DRAM caches are orthogonal to the proposals of this article.

Data Placement in Hybrid Memory. Application data placement in hybrid memory systems as well as NUMA systems has been well studied. Hybrid memory placement policies attempt to fully utilize total system bandwidth by distributing pages between system memory and stacked memory based on the bandwidth ratio (Agarwal et al. 2015; Chou et al. 2015a). NUMA aware placement on the other hand focuses on data placement near computing resources to minimize overall latency (Dashti et al. 2013; Verghese et al. 1996; Bolosky et al. 1989). Our work is orthogonal these proposals.

Address Translation Support for GPUs. Efficient and high performing address translation on GPUs is an important area of research. Recent studies (Power et al. 2014; Pichai et al. 2014) show that simply extending CPU-style TLBs and page walkers to GPUs do not perform well. Consequently, novel mechanisms such as highly threaded page walkers (Power et al. 2014) and intelligent page walk scheduling (Pichai et al. 2014) have been proposed. Our work focuses on improving GPU TLB coverage by allowing the GPU LLC to hold TLB entries. Additionally, we improve GPU TLB miss latency by architecting TLBs in DRAM.

Alternate Page Table Implementation. To reduce memory accesses, *Inverted Page Tables* (Jacob and Mudge 1998; Rashid et al. 1987) by indexing the page table using a hash of the virtual address.

However, the number of memory accesses depends on the frequency of hash collisions. Inverted page tables also do not support mapping of two different virtual addresses to the same physical address (Jacob and Mudge 1998). DRAM-TLBs provide benefits of inverted page tables (i.e., address translation in a single memory access) while retaining the benefits of hierarchical page tables.

9 SUMMARY

Increasing application working-set sizes and growing on-die thread-level parallelism has made TLB performance an important performance bottleneck in today's systems. Even page walk caching mechanisms are unable to accommodate emerging application working-set sizes. As a result, LLT coverage can be very important to overall application performance. However, LLT misses suffer long latency due to multiple memory accesses to the page table. As such, when LLT misses occur, reducing address translation latency is necessary to maintain high performance.

This article proposes low-cost hardware mechanisms to accelerate virtual to physical address translation by improving on-die *LLT coverage* and *LLT miss penalty*. We improve on-die TLB coverage by proposing *Unified Cache and TLB (UCAT)*, which enables the conventional unified LLC to also hold TLB entries. UCAT increases on-die LLT coverage by allowing as many TLB entries as there are cache lines in the conventional on-chip LLC. We show that UCAT improves performance by 60% (up to 4×) on average across a set of memory intensive GPU workloads. These improvements can be realized with negligible changes to the existing LLC architecture.

While UCAT improves performance significantly, it does not decrease the number of page table accesses on an LLT miss. To address this problem we propose embedding TLBs in DRAM (DRAM-TLB). DRAM-TLB serves as the next larger level in the TLB hierarchy architected using DRAM technology. DRAM-TLB can be arbitrarily sized to provide the desired TLB coverage. In steady state, DRAM-TLB can avoid multiple page table accesses and provides address translation with a single memory access. Consequently, DRAM-TLB is a low latency alternative to walking the page table on an LLT miss. Our studies show that DRAM-TLB architected using stacked memory technology improves performance by 22% on average (up to 2.25×). We show that DRAM-TLB requires less than 1% the capacity of our baseline 16GB stacked memory system.

Finally, we propose *DUCATI*, an address translation architecture that combines the benefits of DRAM-TLBs and UCAT while requiring negligible hardware overhead. DUCATI improves performance by 81% (up to 4.5×) for 4KB pages and by 56% and 8% when applied to 64KB and 2MB page systems, respectively.

REFERENCES

Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. 2015. Page placement strategies for GPUs within heterogeneous memory systems. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15).*

ATS. 2009. PCI Express, Address Translation Service. Retrieved from http://composter.com.ua/documents/ats_r1.1_26Jan09.pdf.

Thomas W. Barr, Alan Cox, and Scott Rixner. 2010. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10).*

Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A mechanism for speculative address translation. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 307–318.

Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture.*

Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII).* ACM, New York, NY, 26–35. DOI: https://doi.org/10.1145/1346281.1346286

Abhishek Bhattacharjee. 2013. Large-reach memory management unit caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46).* ACM, New York, NY, 383–394. DOI: https://doi.org/10.1145/2540708.2540741

Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared last-level TLBs for chip multiprocessors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*. IEEE Computer Society, Los Alamitos, CA, 62–63.

Abhishek Bhattacharjee and Margaret Martonosi. 2010. Inter-core cooperative TLB for chip multiprocessors. In *Proceedings of the 15th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, 359–370. DOI: https://doi.org/10.1145/1736020.1736060

W. Bolosky, R. Fitzgerald, and M. Scott. 1989. Simple but effective techniques for NUMA memory management. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP'89)*. ACM, New York, NY, 19–31. DOI: https://doi.org/10.1145/74850.74854

J. Bradley Chen, Anita Borg, and Norman P. Jouppi. 1992. A simulation based study of TLB performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*. ACM, New York, NY, 114–123. DOI: https://doi.org/10.1145/139669.139708

Chiachen Chou, Aamer Jaleel, and Moin Qureshi. 2015b. BEAR: Techniques for mitigating bandwidth bloat in gigascale DRAM caches. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*.

Chiachen Chou, Aamer Jaleel, and Moin K. Qureshi. 2014. CAMEO: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO47)*.

Chiachen Chou, Aamer Jaleel, and M K Qureshi. 2015a. BATMAN: Maximizing bandwidth utilization for hybrid memory systems. *Technical Report for Computer ARchitecture and Emerging Technologies (CARET) Lab, TR-CARET-2015-01.*

CORAL. 2014. *CORAL Procurement Benchmarks*. Retrieved from https://asc.llnl.gov/CORAL-benchmarks/.

Jonathan Corbet. 2017. Five-level page tables. Retrieved from https://lwn.net/Articles/717293/.

Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic management: A holistic approach to memory placement on NUMA systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, NY, 381–394. DOI: https://doi.org/10.1145/2451116.2451157

Alejandro Duran and Michael Klemm. 2012. The intel® many integrated core architecture. In *Proceedings of the 2012 International Conference on High Performance Computing and Simulation (HPCS'12)*. IEEE, 365–366.

John Feehrer, Sumti Jairath, Paul Loewenstein, Ram Sivaramakrishnan, David Smentek, Sebastian Turullols, and Ali Vahidsafa. 2013. The oracle sparc T5 16-core processor scales to eight sockets. *IEEE Micro* 33, 2 (2013), 48–57.

Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large pages may be harmful on NUMA systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, 231–242. http://dl.acm.org/citation.cfm?id=2643634.2643659

NVIDIA GP100. 2016. P100 GPU Accelerator.

Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. 2009. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574* 3 (2009).

HMC Specification 1.0. Retrieved from http://www.hybridmemorycube.org, 2013.

HSA Foundation 2014. *HSA Platform System Architecture Specification*. HSA Foundation. Retrieved from http://www.slideshare.net/hsafoundation/hsa-platform-system-architecture-specification-provisional-verl-10-ratifed.

Intel. 2009. Intel 64 and IA-32 Architectures Optimization Reference Manual.

Bruce Jacob and Trevor Mudge. 1998. Virtual memory: Issues of implementation. *IEEE Comput.* 31, 6 (Jun. 1998), 33–43.

Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. 12. DOI: https://doi.org/10.1145/1815961.1815971

JEDEC. 2013a. *DDR4 SPEC (JESD79-4)*. JEDEC.

JEDEC. 2013b. *High Bandwidth Memory (HBM) DRAM (JESD235).*

James Jeffers, James Reinders, and Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann.

D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi. 2014. Unison cache: A scalable and effective die-stacked DRAM cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 25–37. DOI: https://doi.org/10.1109/MICRO.2014.51

Zhipeng Jiang, Xiaodong Hu, and Suixiang Gao. 2013. A parallel ford-fulkerson algorithm for maximum flow problem. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'13).*

Daniel A. Jiménez. 2013. Insertion and promotion for tree-based pseudolru last-level caches. In *Proceedings of the 46th Annual International Symposium on Microarchitecture*. 13. DOI: https://doi.org/10.1145/2540708.2540733

Stephen Junkins. 2015. The compute architecture of intel processor graphics gen9. *Intel Whitepaper v1* (2015).

Gokul B. Kandiraju and Anand Sivasubramaniam. 2002. Going the distance for TLB prefetching: An application-driven study. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*. IEEE Computer Society, Washington, DC, USA, 195–206. http://dl.acm.org/citation.cfm?id=545215.545237

Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. 2011. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO44)*. ACM, New York, NY, 24–35. DOI: https://doi.org/10.1145/2155620.2155624

Samira M. Khan, Daniel A. JimÃ©nez, and Doug Burgerand Babak Falsafi. 2010. Using dead blocks as a virtual victim cache. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT-19)*.

Milind A. Kulkarni, Martin A. Burtscher, Calin Cascaval, and Keshav Pingali. 2009. Lonestar: A Suite of Parallel Irregular Programs?

Jaekyu Lee and Hyesoon Kim. 2012. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *Proceedings of the 2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA'12)*. IEEE, 1–12.

Gabriel H. Loh and Mark D. Hill. 2011. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *Proceedings of the 44th Annual International Symposium on Microarchitecture*. 11. DOI: https://doi.org/10.1145/2155620.2155673

Piotr R. Luszczek, David H. Bailey, Jack J. Dongarra, Jeremy Kepner, Robert F. Lucas, Rolf Rabenseifner, and Daisuke Takahashi. 2006. The HPC challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. 213.

Joe Macri. 2015. AMD's next generation GPU and high bandwidth memory architecture: FURY. In *Proceedings of the 2015 IEEE Hot Chips 27 Symposium (HCS'15)*. IEEE, 1–26.

Khalid Moammer. 2016. *AMD Zen Raven Ridge APU Features HBM, 128GB/s of Bandwidth and Large GPU.*

Dan Negrut, Radu Serban, Ang Li, and Andrew Seidl. 2014. Unified memory in cuda 6.0. a brief overview of related data access and transfer issues. *Tech. Rep. TR-2014–09, University of Wisconsin—Madison.*

Binh Pham, Arup Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE.

Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced large-reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Los Alamitos, CA, 258–269.

Binh Pham, Jan Vesely, Gabriel Loh, and Abhishek Bhattacharjee. 2015. Large pages and lightweight memory management in virtualized systems: Can you have it both ways? In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. 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 (ASPLOS'14)*. ACM, New York, NY, 16.

Jonathan Power, Mark D. Hill, David Wood, et al. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE, 568–578.

Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. 11. DOI: https://doi.org/10.1145/1250662.1250709

Moinuddin K. Qureshi and Gabe H. Loh. 2012. Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-Tags with a simple and practical design. In *Proceedings of the 2012 45th Annual International Symposium on Microarchitecture*. 12. DOI: https://doi.org/10.1109/MICRO.2012.30

Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. 1988. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers* 37, 8 (1988), 896–908.

Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. 2017. Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 469–480.

Ashley Saulsbury, Fredrik Dahlgren, and Per Stenstrom. 2000. Recency-based TLB preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. ACM, New York, NY, 117–127.

Jaewoong Sim, Gabriel H. Loh, Hyesoon Kim, Mike O'Connor, and Mithuna Thottethodi. 2012. A mostly-clean DRAM cache for effective hit speculation and self-balancing dispatch. In *Proceedings of the 2012 45th Annual International Symposium on Microarchitecture*. 11. DOI: https://doi.org/10.1109/MICRO.2012.31

Jaewoong Sim, Gabriel H. Loh, Vilas Sridharan, and Mike O'Connor. 2013. Resilient die-stacked DRAM caches. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 416–427.

Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights landing: Second-generation intel xeon phi product. *IEEE Micro* 36, 2 (2016), 34–46.

Madhusudhan Talluri and Mark D. Hill. 1994. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*. ACM, New York, NY, 171–182. DOI: https://doi.org/10.1145/195473.195531

Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. 1992. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*. ACM, New York, NY, 415–424. DOI: https://doi.org/10.1145/139669.140406

Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*. ACM, New York, NY, 279–289.

Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. 2016. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *Proceedings of the 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'16)*. IEEE, 161–171.

David A. Wood, Susan J. Eggers, Garth Gibson, Mark D. Hill, and Joan M. Pendleton. 1986. An in-cache address translation mechanism. In *ACM SIGARCH Computer Architecture News*, Vol. 14. IEEE Computer Society Press, 358–365.

Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Jr. Simon C. Steely, and Joel Emer. 2011. SHiP: Signature-based hit predictor for high performance caching. In *Proceedings of the 2012 45th Annual International Symposium on Microarchitecture (Micro-44)*.

Vinson Young, Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. 2018a. ACCORD: Enabling associativity for gigascale DRAM caches by coordinating way-install and way-prediction. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, 328–339.

Vinson Young, Aamer Jaleel, Evgeny Bolotin, Eiman Ebrahimi, David Nellans, and Oreste Villa. 2018b. Combining HW/SW mechanisms to improve NUMA performance of multi-GPU systems. In *Proceedings of the 2018 IEEE 51st International Symposium on Microarchitecture (MICRO51)*. IEEE.

Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. 2016. Towards high performance paged memory for GPUs.

Received February 2018; revised January 2019; accepted January 2019