HPE: Hierarchical Page Eviction Policy for Unified Memory in GPUs

Qi Yu, Bruce Childers, *Member, IEEE*, Libo Huang, Cheng Qian, and Zhiying Wang

***Abstract*—Recent support for unified memory and demand paging has improved graphics processing unit (GPU) programmability and enabled memory oversubscription. However, this support introduces high overhead when page faults occur. Therefore, when the GPU memory fills to capacity, an important issue is how to select eviction candidates. The widely used policy, LRU, and the advanced replacement policies, RRIP and CLOCK-Pro, suffer from inefficiency when dealing with thrashing access patterns. They also incur significant overhead due to managing metadata at page level. In this article, we propose hierarchical page eviction (HPE), a new replacement policy for GPUs with unified memory. Aided by page walk hit information, HPE manages a page set chain dynamically. It uses statistics to classify applications into three categories and selects an appropriate eviction strategy for each category. It also applies dynamic adjustment to switch the eviction strategy when necessary. The simulation results show that, on average, HPE achieves 1.34× and 1.16× speedup (up to 2.81×) over LRU when the oversubscription rate is 75% and 50%, respectively. HPE also outperforms RRIP and CLOCK-Pro.**

***Index Terms*—Access pattern, eviction policy, graphics processing units (GPUs), unified memory.**

I. Introduction

DUE TO massive thread-level parallelism, graphics processing units (GPUs) have been adopted in a wide range of domains, such as graph analytics, machine learning, computational finance, climate modeling, and multimedia [1]. Currently, there are two basic kinds of GPUs in the marketplace: 1) on-die GPUs and 2) discrete GPUs. Although integrated on-die GPUs are widespread, the market is still dominated by PCIe-attached discrete GPUs [2].

In recent years, GPUs have improved significantly in terms of the performance and programmability. The performance improvement relies on hardware innovation and increased compute density [3]. Improved programmability is a result of changes in programming models, such as CUDA [4] and OpenCL [5]. In previous programming models, programmers were responsible for GPU memory management and CPU-GPU data transfer. For example, to process data on a GPU, a programmer had to allocate memory in both the CPU and GPU, copy input data from CPU to GPU, and finally copy output data from GPU to CPU with functions provided by the programming model. This approach has three shortcomings: 1) it puts heavy burden on the programmer; 2) it uses complicated asynchronous user-directed constructs to overlap kernel execution and CPU-GPU data transfer [6]; and, 3) it does not support memory *oversubscription*, i.e., if the application footprint exceeds the GPU memory capacity, programmers have to carefully divide the datasets into pieces and move these pieces manually, which is error-prone and time-consuming.

To simplify this process, GPU vendors have started to support *memory virtualization*. For instance, NVIDIA introduced the unified memory [7] and the heterogeneous system architecture (HSA) foundation introduced shared virtual memory [8]. In recent CUDA/OpenCL versions, demand paging [9] is further supported, which relies on a software-managed runtime to automatically page memory in and out of the GPU on demand. With on-demand page migration, programmers are unburdened from explicitly manipulating memory and data transfer. Demand paging also makes the overlap between GPU execution and CPU-GPU data transfer user-transparent because, in this situation, the GPU can begin execution before any data transfer has occurred. Another benefit is oversubscription; that is, the GPU can compute across datasets that exceed the GPU memory capacity. Unfortunately, the benefits of demand paging come at a cost. As GPUs do not support context switching to operating system service routines, the manipulation of the GPU page table and page transfers is offloaded to a software runtime on the host CPU [10]. When a page fault occurs, several PCIe round trips and interaction with the host CPU are required, incurring significant overhead [11]. Therefore, when the GPU memory is full, how to select eviction candidates is an important issue.

To better understand the limitations of commonly used eviction policies, we study the access patterns of GPU applications from Rodinia [12], Parboil [13], and Polybench [14]. We make three observations. First, we find six typical access patterns, among which, two are simple patterns and four are complex ones. Second, we find that virtual pages with continuous addresses have good spatial locality (in terms of access frequency and occasion). Third, we observe that virtual pages are usually hot for a short duration after faulting;

Manuscript received March 21, 2019; revised June 28, 2019 and September 5, 2019; accepted September 17, 2019. Date of publication October 4, 2019; date of current version September 18, 2020. This work was supported in part by the National Natural Science Foundation of China under Grant 61433019, Grant 61872374, and Grant 61572508. This article was recommended by Associate Editor Z. Shao. *(Corresponding author: Libo Huang.)*

Q. Yu, L. Huang, C. Qian, and Z. Wang are with the School of Computer, National University of Defense Technology, Changsha 410073, China (e-mail: yuqi13@nudt.edu.cn; libohuang@nudt.edu.cn; qiancheng@nudt.edu.cn; zywang@nudt.edu.cn).

B. Childers is with the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260 USA (e-mail: childers@pitt.edu).

Digital Object Identifier 10.1109/TCAD.2019.2944790

that is, pages are re-referenced in the near future after migration. Consequently, immediately evicting newly touched pages causes thrashing; we call this phenomenon "instant thrashing."

We propose a new page eviction policy, *hierarchical page eviction* (HPE), to address this issue. Our policy maintains a software-managed page set chain and selects eviction candidates according to recency and frequency of page sets. The simulation results show that for 23 selected applications with six access patterns, HPE achieves an average speedup of 1.34× and 1.16× over LRU when the oversubscription rate is 75% and 50%, respectively. HPE also outperforms two advanced replacement policies, RRIP [15] and CLOCK-Pro [16].

This article makes the following contributions. **First**, we analyze the access patterns of GPU applications from three representative benchmark suites. **Second**, we propose HPE. With a software-managed page set chain, HPE aims to address LRU's inefficiency for thrashing access patterns, while preserving LRU's advantages for LRU-friendly patterns. To achieve this goal, HPE uses statistics to classify applications into three categories and selects an eviction strategy for each category. It also adjusts the eviction strategy when necessary. **Finally**, we present a comprehensive sensitivity test, evaluation, and overhead analysis.

The rest of this article is organized as follows. Section II introduces background on unified memory. Section III describes our experimental setup and motivates the need for a new page eviction policy. Section IV presents design details of HPE. Section V evaluates HPE and compares it against other eviction policies. Section VI discusses related work and Section VII concludes this article.

II. Background

Recent introduction of unified memory in a GPU requires support for address translation and demand paging. Although the details of memory hierarchy design for widely used GPU architectures from NVIDIA, AMD, and Intel have not been published, it is accepted that contemporary GPUs support TLB-based address translation [11]. Although we use NVIDIA's terminology to describe GPU architecture, HPE is broadly applicable to other GPUs, such as the AMD GPU architecture. Recent research [11], [17] presents two variants of address translation designs. First, a shared page walk cache is used in the page table walker [10], [17]. Second, a shared L2 TLB is used after private L1 TLBs [11], [18], [19]. In this article, we adopt the second design due to better performance than the first one [11]. The second design's structure is shown in Fig. 1.

Each *streaming multiprocessor* (SM) has a private L1 TLB shared across all CUDA cores. These per-SM L1 TLBs are backed up by a larger L2 TLB shared by all SMs. A translation request that misses in the TLB hierarchy is sent to the GPU memory management unit (MMU) [10], [11], [17], [20], [21]. The page table walker accesses each level of the page table to retrieve the desired data from either the shared L2 cache or main memory.

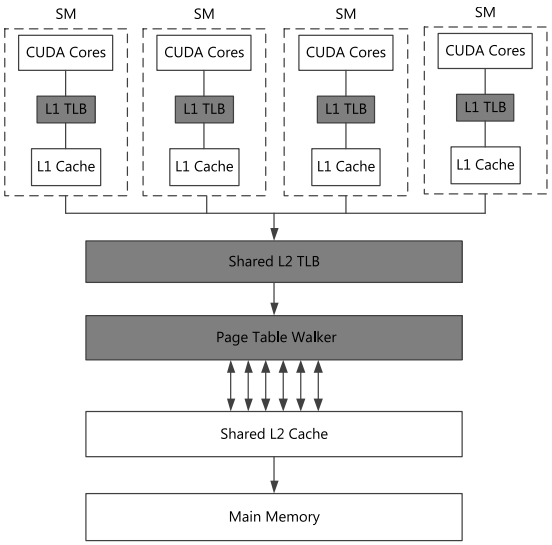


Fig. 1. Baseline GPU architecture with support for address translation.

If the page table walker fails to find a desired address mapping in the page table, a page fault occurs. As GPUs themselves cannot handle page faults in the execution pipeline (as CPUs typically do today) [22], GPU vendors (NVIDIA and AMD) use a software runtime (executes on the host CPU) to handle page faults [23]. In this case, the GPU first stops address translation in the faulted SM, and then sends a request to the CPU. The OS on the CPU performs a page table lookup and migrates the faulted page to the GPU. If the GPU memory fills to capacity, before serving the page fault, the GPU driver selects an eviction candidate to page out of the GPU memory. Then the GPU transfers the evicted page to the CPU memory via PCIe. When the page fault is serviced, the GPU retries the address translation for the faulting address. Page fault handling requires several PCIe round trips and interaction with the host CPU; and thus, it incurs high overhead, i.e., more than 20 microseconds [10], [24].

III. Motivation

In this section, we experimentally motivate the need for a new page eviction policy. We extend GPGPU-Sim 3.2.2 [25] with TLBs and a GMMU (GPU MMU) to support a basic infrastructure of unified memory. We adopt a two-level TLB design, including a private L1 TLB for each SM and an L2 TLB shared by all SMs, which is similar to prior research [11], [18], [19], [21]. A single-level page table and a fixed page walk latency (eight cycles) are used to simplify simulation. We also use a fixed latency (20 μs, same as [10]) to represent page fault handling and eviction. We implemented the replayable far-fault mechanism [10] to enable each SM to continue execution in the presence of page faults. We choose 4-KB OS pages, which is same as prior work [10], [17], [19], [20]. This is the default page size in current GPUs [26]. The system configuration is shown in Table I.

We use selected applications from Rodinia, Parboil, and Polybench for evaluation, whose details are shown in Table II.

$(a_1,\ a_2,\ a_3,\ \dots,\ a_k)$ **I: Streaming Access Pattern** $(k = \infty)$	$(a_1, a_2, a_3, \dots, a_k)^N$ **II: Thrashing Access Pattern** $(k >$ **memory size**; $N \geq 2)$
$(a_1^{N_1} \cdot \varepsilon_1, a_2^{N_2} \cdot \varepsilon_2, \dots, a_k^{N_k} \cdot \varepsilon_k)$ **III: Part Repetitive Access Pattern** $(1 \leq N_1, N_2, \dots, N_k; 0 < \varepsilon_1, \varepsilon_2, \dots, \varepsilon_k < 1)$	$(a_1^{N_1}, a_2^{N_2}, a_3^{N_3}, \dots, a_k^{N_k})$ **IV: Most Repetitive Access Pattern** $(1 \leq N_1, N_2, \dots, N_k)$
$(a_1^{N_1}, a_2^{N_2}, a_3^{N_3}, \dots, a_k^{N_k})^N$ **V: Repetitive-Thrashing Access Pattern** $(k >$ **memory size**; $N \geq 2$; $1 \leq N_1, N_2, \dots, N_k)$	$(a_1^{N_1}, \dots, a_{k1}^{N_{k1}}), (a_{k1+1}^{N_{k1+1}}, \dots, a_{k2}^{N_{k2}}), \dots, (a_{kn+1}^{N_{kn+1}}, \dots, a_k^{N_k})$ **VI: Region Moving Access Pattern** $(k1 < k2 < \cdots < kn < k; 1 \leq N_1, N_2, \dots, N_{k1+1}, \dots, N_k)$

Fig. 2. Representative access patterns found in selected GPU workloads (Types I and II refer to [15]).

TABLE I
CONFIGURATION OF THE SIMULATED SYSTEM

GPU Arch.	NVIDIA GTX-480 Fermi-like
GPU Cores	15 cores, 1.4GHz
Private L1 cache	16KB, 4-way associative, LRU
Private L1 TLB	128-entry per SM, single port, 1-cycle latency, LRU, support hit under miss
Shared L2 cache	1.5MB total, 128KB/DRAM channel, 8-way associative, LRU
Shared L2 TLB	512-entry, 16-associative, LRU, 10-cycle latency, 2 ports
DRAM	GDDR5, 12-channel, FR-FCFS scheduler, 177GB/s aggregate
CPU-GPU interconnect	16GB/s, 20 μs page fault service time

TABLE II
WORKLOAD CHARACTERISTICS

access pattern	benchmark suite	application and Abbr.
Type I	Rodinia	hotspot (HOT), leukocyte (LEU)
	Parboil	cutcp (CUT)
	Polybench	2DCONV (2DC), GEMM (GEM)
Type II	Rodinia	srad_v2 (SRD), hotspot3D (HSD)
	Parboil	mri-q (MRQ), stencil (STN)
Type III	Rodinia	pathfinder (PAT), dwt2d (DWT), backprop (BKP), kmeans (KMN)
	Parboil	sad (SAD)
Type IV	Rodinia	nw (NW), bfs (BFS)
	Polybench	MVT (MVT)
Type V	Rodinia	heartwall (HWL)
	Parboil	sgemm (SGM), histo (HIS), spmv (SPV)
Type VI	Rodinia	b+tree (B+T), hybridsort (HYB)

The footprints of these applications vary from 3 MB to 130 MB with an average of 37 MB. Long simulation time prevents us from evaluating applications with larger footprints. We elided some applications due to certain reasons.

1) *Too Small Footprint: myocyte, lud, particlefilter* from Rodinia and *mri-q* (*small*), *sgemm* (*small*), *bfs* (*UT, NY*), *spmv* (*small, medium*), *tpacf* (*small*) from Parboil.
2) *Too Long Simulation Time: streamcluster* from Rodinia, *sad* (*large*), *stencil* (*default*), *bfs* (*1M*), *lbm* (*short*), *mri-gridding*, *cutcp* (*large*), *tpacf* (*medium*) from Parboil and *SYRK, SYR2K, CORR, COVAR, 2MM, BICG* from Polybench.
3) *Runtime Error: cfd, huffman, mummergpu, nn* from Rodinia, *lbm* (*long*), *histo* (*large*), *tpacf* (*large*) from Parboil and *FDTD-2D, GRAMSCHM* from Polybench.
4) *Duplicated Access Patterns With Other Selected Applications: 3DCONV* (with *2DCONV*), and *ATAX, GESUMMV* (with *MVT*).

A. Access Pattern Types

Different GPU workloads have different access patterns. To better understand when existing eviction policies (mentioned in Section I) behave well or poorly, we studied application access patterns from Rodinia, Parboil, and Polybench. We find six representative patterns, which are shown in Fig. 2.

In this figure, a_i denotes a virtual page; $a_i^{N_i}$ means a_i is referenced N_i times (refer to frequency, the same below). $(a_1, a_2, \dots, a_k)$ denotes a temporal sequence of references to k unique virtual pages and $(a_1^{N_1}, a_2^{N_2}, \dots, a_k^{N_k})$ represents a temporal sequence with different reference frequencies. A temporal sequence repeating N times with the same frequency and with different frequencies is denoted as $(a_1, a_2, \dots, a_k)^N$ and $(a_1^{N_1}, a_2^{N_2}, \dots, a_k^{N_k})^N$, respectively.

Types I and II are simple access patterns, where all pages are referenced the same number of times (1 or N). However, there are complex access patterns in which virtual pages are referenced a different number of times and with different probability, such as types III–VI. Type III is a temporal sequence where parts of virtual pages are referenced multiple times with some probability. Type IV represents a temporal sequence in which most virtual pages are referenced multiple times. Type V is a combination of types II and IV: a temporal sequence repeats N times; in each iteration, most virtual pages are referenced multiple times. In type VI, virtual pages are divided into kn address regions, and in each region, pages are referenced multiple times for a certain duration of time. The application then continues to access pages in the next region. For types III–VI, different page references usually intersect with each other (in terms of reference order).

B. Limitations of LRU and RRIP

From Fig. 2, we can infer that LRU performs well for types I and VI, but poorly for type II. For this type, LRU fails to preserve some of the working set (WS) in the GPU memory. RRIP is expected to perform well for type II; however, due to instant thrashing, RRIP has limited speedup over LRU. For types III–V, it is difficult to determine whether these

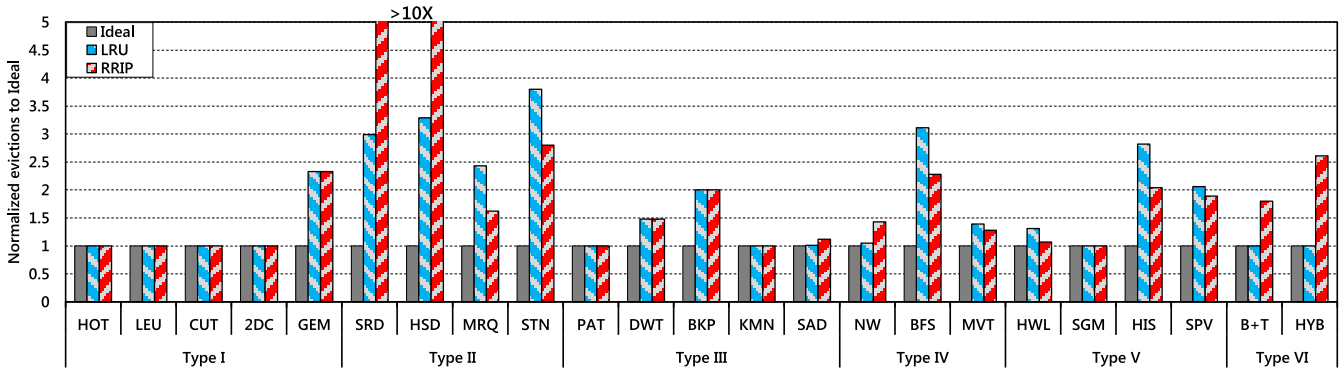


Fig. 3. Evictions of LRU and RRIP normalized to the ideal policy.

eviction policies perform well or poorly, because performance is influenced by several factors, such as which pages are referenced multiple times and when the pages are referenced. To show the limitations of LRU and RRIP, we conducted simulations under an oversubscription rate of 75%, which means only 75% of application footprint fits in the GPU memory. As a baseline, we use an offline eviction policy to explore the upper bound of performance, which is similar to Belady's MIN algorithm [27]. We call this policy "Ideal." We normalize evictions of LRU and RRIP to Ideal. Fig. 3 shows the result. We make three observations. First, for type II, RRIP incurs significant thrashing for *SRD* and *HSD*. Despite outperforming LRU for *MRQ* and *STN*, RRIP evicts 50% more pages than Ideal. Second, LRU performs well for type I (except for *GEM*) and type VI, while RRIP performs poorly for type VI. Third, for types III–V, it is not easy to determine which policy performs better; however, both policies perform poorly for some applications of types IV and V, such as *BFS*, *HIS*, and *SPV*.

We also observe that virtual pages with continuous addresses have good spatial locality: these pages have similar reference frequency and occasion. However, LRU and RRIP fail to recognize this behavior. When several virtual pages from different address regions are referenced on a similar occasion, LRU and RRIP evict them in order (according to recency or frequency), which may evict some pages that are needed in the future. Finally, for the GPU memory with hundreds of thousands of pages, managing an LRU/RRIP chain at page level is rather expensive. In summary, LRU and RRIP have three disadvantages when applied to unified memory. First, they are ineffective to deal with thrashing access patterns. Second, they fail to recognize spatial locality among continuous virtual pages. Third, management overhead is high. Therefore, for GPUs supporting unified memory, an eviction policy with higher accuracy and lower overhead is necessary.

IV. Hierarchical Page Eviction Policy

Based on the analysis in Section III, we propose HPE. For convenience, we define two concepts.

1) *Page Set:* A group of virtual pages with continuous addresses (like "chunk" in NVIDIA Pascal or later GPUs [28]). For example, page set 8000 with a size of 16 constitutes virtual pages that have the following addresses: 80000, 80001, . . . , 8000f.
2) *Interval:* A fixed partition unit of the running process, which is equal to a specified number of page faults.

HPE addresses the issues of LRU and RRIP as follows. First, to alleviate instant thrashing, HPE does not evict newly touched page sets immediately; instead, it waits until these page sets become "cold." Second, a page set is composed of certain continuous virtual pages, and thus, the spatial locality among these pages is exploited. Finally, managing a page set chain at the page set level (rather than at the page level) reduces chain length as well as management overhead.

A. Main Idea

For GPUs supporting unified memory, the eviction policy is managed by the GPU driver, which is only invoked on page faults [10]. In other words, the eviction policy sees only part of references to the page table in the GPU memory. Due to the absence of page walk hit information, a policy may make the wrong decisions in some cases. We address this issue by enhancing HPE with page walk hit information. Fig. 4 depicts the overall architecture of HPE (shaded components). It has two parts: a set-associative cache (❶) on the GPU side and a software mechanism (❷) on the CPU side. The set-associative cache is called "hit information record cache" (HIR for short). It is placed beside the page table walker and records page walk hit information. The information in HIR is first transferred to a buffer in the GPU memory then to the GPU driver periodically (in terms of number of page faults) via PCIe to update the page set chain.

On the CPU side, we enhance existing GPU driver with a software mechanism. The mechanism has three parts: 1) a page set chain (**A**); 2) the global mechanism (**B**); and 3) the dynamic adjustment (**C**). When the GPU memory fills to capacity, the global mechanism selects an eviction strategy according to the classification result. To deal with wrong classifications and variance in access behavior, the dynamic adjustment switches the eviction strategy when necessary. When an eviction is required, a page set is selected first, and then its virtual pages are selected in address order, and corresponding physical pages are evicted to the CPU memory. Note, only one physical page is evicted each time.

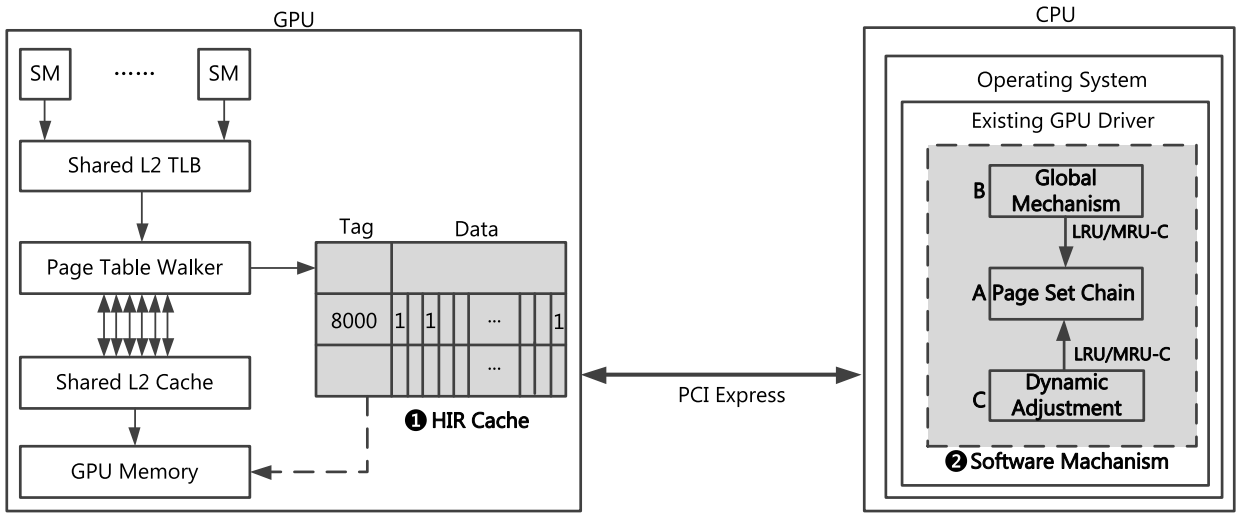


Fig. 4. Overall architecture of HPE.

B. HIR Cache

The structure and location of HIR are shown in Fig. 4. Each entry in HIR has two fields: tag and data. The tag is page set address and data is a counter vector, which records the number of references to each page in a page set. When a page walk request is made, the page table walker begins a walk. Once the walker knows that the request is a hit, it notifies HIR with the page address. Then the page set address and offset are calculated. HIR is searched with the page set address, and the corresponding counter is increased according to the offset.

The hit information is transferred to the GPU driver very *n*th page fault (rather than every page fault). We do this for two reasons. First, HPE does not require an exact reference order to maintain the page set chain. Second, for some applications, there are few page walk hits between consecutive page faults. Therefore, transferring more information in a less frequent way reduces interruption of the GPU driver. Before evicting a page, the content of HIR is copied to a buffer in the GPU memory. The information is then transferred to the GPU driver along with evicted page via PCIe. When this process completes, HIR is flushed. Note, only touched entries are transferred.

This design is better than using a buffer to record page addresses in order, because in some cases, the latter method may incur significant storage and transfer overhead due to GPU's burst access behavior [29]. On average, compared to recording the page addresses with a buffer, HIR reduces storage cost by 63% and 53% when the oversubscription rate is 75% and 50%, respectively. However, HIR has two issues: 1) the reference order is lost and 2) way conflicts may exist, and consequently, some pages' information may be lost. To deal with the first issue, we use a vector to record the first touch order of entries in HIR. The touched entries are copied to the buffer according to the content of this vector. By keeping a relaxed reference order, the impact of losing order information is reduced. For the second issue, we find that an 8-way associative HIR with 1024 entries avoids way conflicts in the simulations for most applications (except *MVT*).

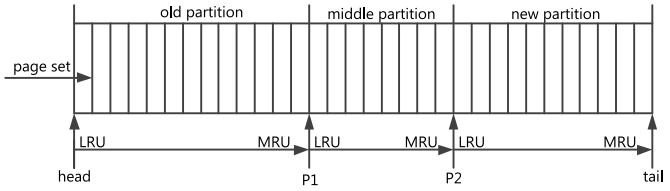


Fig. 5. Structure of the page set chain.

C. Page Set Chain: Structure and Operation

The structure of the page set chain is shown in Fig. 5. The page set chain has three partitions according to the recency of page sets. *Old partition* (between head and P1) contains page sets that were previously referenced but have not been referenced in the last and current intervals. *Middle partition* (between P1 and P2) contains page sets that were referenced in the last interval. *New partition* (between P2 and tail) contains page sets that are referenced in the current interval. The boundaries between the partitions are managed by two pointers (P1 and P2 in the figure). Each page set has an entry in the chain, and the entry has four fields: 1) a tag; 2) a saturating counter; 3) a bit vector; and 4) a flag. The tag is the page set address. The counter records the number of touches to a page set and saturates at 64 (i.e., once the counter reaches 64, it does not increase anymore even if there are further touches to the page set). We set this restriction for two reasons. First, we find that once the access frequency of page sets exceeds a threshold, the exact value (e.g., 80 or 100) is not important when selecting eviction candidates. Second, the restriction simplifies classification and reduces search overhead. Each page in a page set has a bit in the bit vector. A bit value of "0" indicates that the page has not been touched, and a bit value of "1" indicates the page has been referenced. The bit vector is used to determine whether a page set should be divided (to deal with divergence in pages' access behavior) and the flag indicates whether a page set has been divided (see below for details).

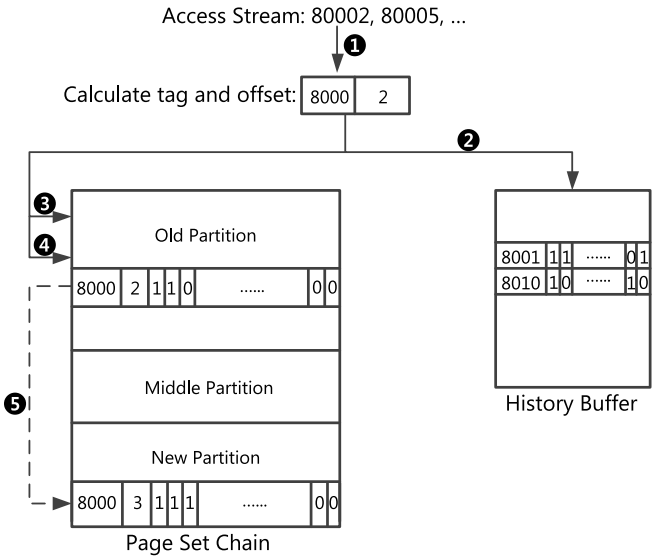


Fig. 6. Operations to update the page set chain.

In most cases, pages in a page set behave similarly. However, in some situations, they may have different behavior. An example is *NW* (from Rodinia). This application touches pages with even addresses (even pages) and odd addresses (odd pages) on different occasions. For example, we assume page set 8000 constitutes 16 pages: 80000–8000f. In the first certain intervals, even pages (80000, 80002, . . . , 8000e) are referenced; after that, odd pages (80001, 80003, . . . , 8000f) are accessed. If a page set is evicted immediately after even pages have been touched, thrashing happens as odd pages are accessed in the near future. We find that managing pages according to even/odd address is not a one-size-fits-all solution.

Instead, HPE dynamically divides a page set into two page sets when necessary. To achieve this, HPE counts the number of bits (in the bit vector) that are set to 1 when the saturating counter reaches 64. If not all bits are set to 1 (i.e., some pages have not been touched), the page set is divided into two parts; otherwise, it remains one page set. A page set is divided according to its bit vector: pages that have been touched stay in the current page set (called "primary") and pages that have not been touched are put into a new page set (called "secondary") if they are touched afterward. The primary and the secondary have different tags. For a page set, there may exist divergence between the first division and subsequent divisions (i.e., the page set is re-referenced after eviction). In this case, the result of the first division is used due to better performance. We use a history buffer to record the metadata (tag and bit vector) of primary when it is removed from the page set chain.

Fig. 6 shows the operations to update the page set chain with hit/faulted pages. First, the tag and offset are calculated from page addresses with right shift and bit operations (❶). The history buffer is searched using the tag and offset (❷). There are three cases: 1) the page set was previously evicted, and the page belongs to primary; 2) the page set was previously evicted, but the page belongs to secondary; and 3) the page set was not previously evicted. If 1) or 3) occurs, the page set chain is searched with the primary tag (❸); otherwise, the secondary tag is used (❹). If a matching entry is found, the counter and bit vector are updated; otherwise, a new entry is created and inserted into the MRU position of the new partition. If the matching entry is in the old or middle partition, it is moved to the MRU position of the new partition (❺). The current interval is over when the number of page faults reaches a threshold. At this point, P1 and P2 are changed to update the partitions. Once all pages in a page set have been evicted, the page set is removed from the page set chain. Note: 1) only page faults update the bit vector and 2) within an interval, once a page set has been placed into the new partition due to insertion or movement (from old partition or middle partition), following touches to this page set will not trigger its movement.

D. Global Mechanism

When the GPU memory fills to capacity, page eviction is required. There are two important issues to consider. One is from which partition to select eviction candidates, and the other is how to select eviction candidates. For the first issue, to alleviate instant thrashing, HPE selects eviction candidates from the old partition first, if it is not empty. However, in rare cases, the old partition may be empty due to eviction and movement. In these cases, HPE selects eviction candidates from the middle partition if it is not empty, or the new partition if the middle partition is empty too. By keeping newly touched page sets in the new partition, and selecting eviction candidates from the old partition, instant thrashing is alleviated.

For the second issue, HPE uses statistics to classify applications into three categories, and then selects an appropriate eviction strategy for each category. For the convenience of description, we define four counters and two variables as follows.

1) *Regular Counter:* A counter divisible by page set size.
2) *Irregular Counter:* A counter indivisible by page set size.
3) *Small and Regular Counter:* A counter whose value is equal to page set size or 2 × page set size.
4) *Large and Regular Counter:* A counter whose value is equal to 3 × page set size or 4 × page set size.
5) $Ratio_1$: The number of page sets that have an irregular counter divided by the number of page sets that have a regular counter.
6) $Ratio_2$: The number of page sets that have a large and regular counter divided by the number of page sets that have a small and regular counter.

In our experiments, when the GPU memory is full for the first time, we counted the number of page sets with different counter types for selected applications. We observe that for applications of types I–III (except *KMN* and *SAD*), 95% of page sets have a small and regular counter; for applications of types IV–VI (except *SGM*), most page sets either have a large and regular counter (91.4%) or have an irregular counter (79.9%).

Based on this observation, we describe the classification in Table III. If most page sets have a small and regular counter

TABLE III
STATISTICS-BASED CLASSIFICATION

category	$ratio_1$	$ratio_2$
regular	$\leq$ threshold	< 2
irregular#1	$\leq$ threshold	≥ 2
irregular#2	$>$ threshold	

($ratio_1 \leq threshold$ and $ratio_2 < 2$), the application is classified as **regular**; if most page sets have a large and regular counter ($ratio_1 \leq threshold$ and $ratio_2 \geq 2$), the application is classified as **irregular#1**. Finally, if most page sets have an irregular counter ($ratio_1 > threshold$), the application is classified as **irregular#2**.

For **regular** applications, HPE uses a strategy that considers both recency and frequency of page sets, which is called MRU-C (MRU-counter based). MRU-C searches from the MRU position of the old partition until a page set that satisfies the following requirements is found: 1) a page set whose counter is equal to page set size or 2) a page set with the minimum counter (least frequently used) if all page set counters are larger than page set size. The page set that satisfies the requirements is the "qualified page set." Searching from the MRU position utilizes recency of page sets, while using page set counters for selection utilizes frequency of page sets.

We use MRU-C for three reasons. First, for **regular** applications, searching from the MRU position achieves better average performance than from the LRU position, especially for applications with thrashing patterns. Second, if not all page sets are fully populated (i.e., all pages in the page set have been touched), evicting a fully populated page set is less likely to incur thrashing than evicting a page set with counter smaller than page set size. On the contrary, if all page set counters are larger than page set size, the page set with the minimum counter is less likely to be re-referenced. Third, this strategy reduces search overhead because most page sets in regular applications have a small and regular counter. For **irregular#1** and **irregular#2** applications, HPE starts with LRU due to better average performance. If the old partition becomes empty, LRU is used to select eviction candidates in the middle partition or new partition.

E. Dynamic Adjustment

Although the classification achieves acceptable results, mistakes may be made in some cases. For instance, *BFS* is classified as **irregular#1**. However, LRU causes severe thrashing because there exists a thrashing pattern in *BFS*'s page walk trace. The access behavior may also change during runtime. To address these issues, HPE dynamically adjusts the eviction strategy when necessary. For this purpose, an FIFO buffer is allocated to LRU and MRU-C, respectively. The buffer stores evicted virtual page addresses in the last two intervals. In addition, each strategy has a separate counter to record the number of wrong evictions (i.e., page fault again after eviction). The counter is reset periodically at the end of each interval. Through sensitivity test (result is not shown), we find that triggering adjustment when the counter reaches page set size can filter most unnecessary switches.

Algorithm 1 Dynamic Adjustment

```
For regular applications with large footprint:
strategy ← MRU-C
record wrong evictions of MRU-C
if wrong_eviction = page set size then
    strategy ← MRU-C
    jump search point forward by 16
end if

For irregular#2 applications:
strategy ← LRU
while program is not over do
    record wrong evictions of strategy
    if wrong_eviction = page set size then
        strategy ← longer_interval(LRU, MRU-C)
    end if
end while
```

Dynamic adjustment is shown in Algorithm 1. For **regular** applications, MRU-C incurs instant thrashing in some cases. For example, *SRD* (from Rodinia) and *STN* (from Parboil) experience instant thrashing when selecting eviction candidates from the MRU position of the old partition. A possible solution is to select "colder" page sets by jumping the search point forward by a certain distance, because page sets from the new search point are "older" in terms of recency. However, this solution may degrade performance of applications with smaller footprints (e.g., *STN*). For these applications, the older page sets usually have a larger counter, and thus, are more likely to be re-referenced. On this basis, for **regular** applications, HPE counts the number of page sets in the old partition when the GPU memory is full for the first time. If the number is smaller than $4 \times$*page set size*, HPE does not adjust the eviction strategy even if the requirement is satisfied. Otherwise, HPE jumps the search point forward by 16 (this distance can alleviate instant thrashing in most cases) and continues selecting candidates from the new point. For **irregular#1** applications, HPE remains with LRU because MRU-C may cause instant thrashing due to bursty page walks. For **irregular#2** applications, HPE adjusts the eviction strategy by comparing the number of intervals a strategy lasts and selects the strategy that is used for a longer time.

F. Discussion

The page set chain is simple to implement in existing GPU driver. Compared to LRU, HPE needs to add the following fields or components: a per page set counter, bit vector, a flag, two pointers for boundary management, a history buffer to store evicted primary page sets and two FIFO buffers for dynamic adjustment.

In addition, several features make HPE's operations low cost. First, the management unit is a page set rather than a page. As a result, the chain length and management overhead are reduced. Second, HIR does not incur significant overhead (see Section V-C for more details). Recording hit information is not on the critical path of a page walk, and updating the page

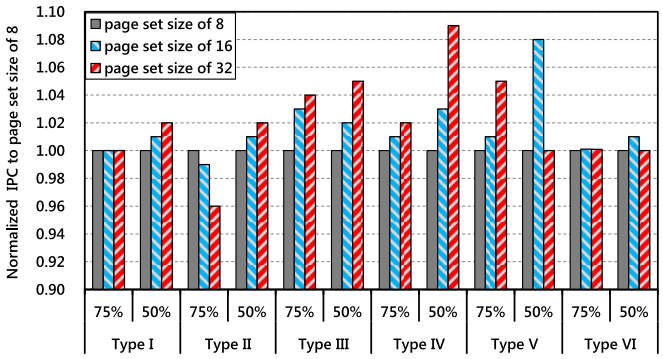


Fig. 7. HPE's sensitivity to page set size.

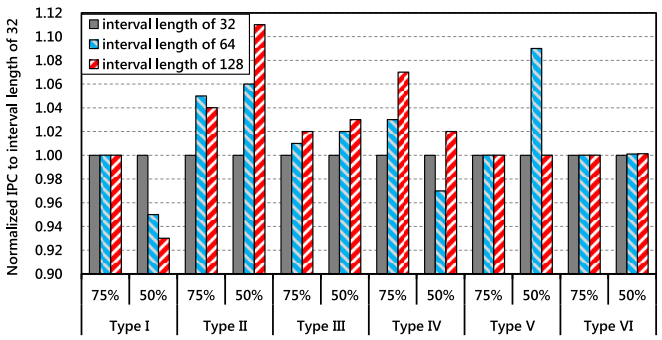


Fig. 8. HPE's sensitivity to interval length.

set chain using hit information is not on the critical path of page fault handling. Third, within an interval, once a page set has been placed into the new partition, subsequent references will not trigger movement of the page set anymore. Therefore, the movement frequency is reduced. Last but not least, only a small portion of selected applications (*NW* and *MVT*) need to divide page sets. For most applications, there is no need to search the history buffer because their buffers are empty.

V. Evaluation

We ran selected applications without constrained memory to get the number of compulsory page faults. We then reduced memory size to two oversubscription rates: 75% and 50%, which means only 75% and 50% of application footprint fits in the GPU memory. To reduce simulation time, we limited the instructions of some applications (*SRD, HSD,* etc.) but kept their access patterns intact. We used an 8-way associative HIR with 1024 entries because this configuration eliminates way conflicts in most cases. We used a 1-cycle lookup latency for HIR. We also compare HPE against LRU, random, RRIP, and CLOCK-Pro.

For HPE, we only focused on performance improvement on the GPU side. We did not evaluate the performance on the CPU side due to minimal performance overhead introduced to existing GPU driver and limitation of experimental platform (GPGPU-Sim). We did not consider the power consumption of the GPU and CPU for two reasons. First, due to small capacity, HIR introduces minimal additional power relative to total GPU power consumption. Second, our simulation infrastructure limits the ability to measure CPU power cost. In addition, we expect HPE introduces small additional power cost to the CPU due to its minimal performance overhead.

A. Sensitivity Study

HPE relies on several parameters to work efficiently: page set size, interval length, the classification threshold, the depth of FIFO buffers, the requirement for switching the eviction strategy, and the interval for transferring page walk hit information. We determined the first three parameters with a sensitivity test, in which we turned off dynamic adjustment and selected an appropriate eviction strategy for each application manually before simulation. In addition, we used an ideal model where page walk hit information is transferred to the GPU driver directly without using HIR. For the last parameter, we turned on dynamic adjustment and enabled HIR.

Page Set Size: To update the page set chain, the tag (page set address) and offset are calculated first. For the purpose of simplifying calculation, we assume the page set size is a value of 2^n. For instance, if the page set size is 16, the tag is calculated by shifting the page address right by 4 bits. On the one hand, the length of page set chain is inversely proportional to page set size. From this point of view, a page set should include more pages to reduce the management overhead. On the other hand, more pages in a page set are more likely to behave differently (in terms of access frequency and occasion). Consequently, evicting a page set with more pages is more likely to incur thrashing.

On this basis, we test three page set sizes: 8, 16, and 32 with interval length of 64. We calculate the average IPC for applications with the same access pattern type, and normalize the result to a page set size of 8. Fig. 7 shows that the three page set sizes have performance difference within 10%. Therefore, setting page set size to 32 seems better considering both chain length and performance. However, we find that in this case, the $ratio_1$ of some **regular** applications is higher, making it difficult for classification. Therefore, we select a less aggressive value of 16.

Interval Length: This parameter has an important impact on HPE's ability to alleviate instant thrashing. On the one hand, a small interval length causes instant thrashing when selecting eviction candidates from the old partition. On the other hand, a large interval length fails to preserve the WS in the GPU memory, especially for applications with type II. We assume the interval length is a multiple of page set size for simplicity. We test three interval lengths: 32, 64, and 128 with a page set size of 16. We calculate the average IPC for applications with the same access pattern type and normalize the result to an interval length of 32.

Fig. 8 shows that the three interval lengths have performance difference within 12%. On average, interval length of 64 and 128 perform slightly better than interval length of 32. However, we find that interval length of 128 performs unstably for applications with type II (i.e., performs best for *SRD*, but performs worst for *STN*). Thus, we set interval length to 64.

Classification Threshold: The statistics method relies on $ratio_1$ and $ratio_2$ to classify applications into three categories.

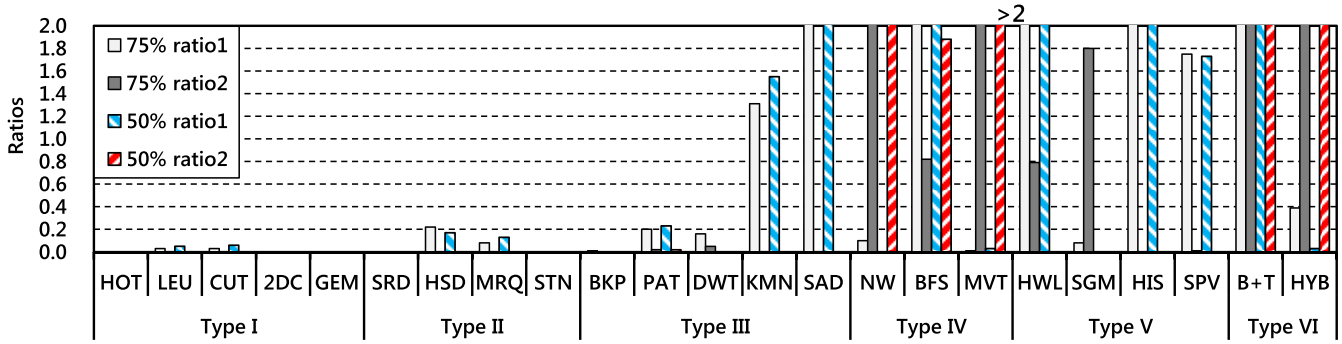


Fig. 9. $ratio_1$ and $ratio_2$ of selected applications.

When the GPU memory fills to capacity for the first time, we calculated these values, as shown in Fig. 9. We see that most applications of types I–III have small (less than 0.3) $ratio_1$ and $ratio_2$, with outliers of *KMN* and *SAD*. Although these two applications belong to type III, they have a large $ratio_1$. We also find that LRU performs better than MRU-C for *KMN* and performs similarly to MRU-C for *SAD*. Therefore, classifying these two applications as **irregular#2** is more appropriate than classifying them as **regular**.

Most applications of types IV–VI either have large $ratio_1$ or large $ratio_2$, with an outlier of *SGM*. This application belongs to type V; however, its $ratio_1$ is rather small. Besides, part of its access pattern is like type II. We find that LRU performs similarly to MRU-C for *SGM*. Therefore, classifying it as **regular** is appropriate. We set the threshold for $ratio_1$ to 0.3 for two reasons. First, this value achieves acceptable classification results. Second, this value satisfies the requirement for regular applications that most page sets have a small and regular counter.

Transfer Interval: To find an appropriate interval for transferring page walk hit information (we call this *transfer interval*) to the GPU driver, we conducted a sensitivity test to evaluate transfer intervals of 1, 8, 16, 32, and 64. We found that 16 makes the best tradeoff between frequency and performance (result is not shown due to space limits). Thus, the page walk hit information is transferred to the GPU driver every 16th page fault.

Other Parameters: As interval length is set to 64, the length of FIFO buffer should be 128 (two intervals). As page set size is set to 16, the threshold for a number of wrong evictions should be 16 as well.

In summary, we set the parameters as follows. Page set size is 16, interval length is 64, threshold for $ratio_1$ is 0.3, the depth of FIFO buffer is 128, threshold for number of wrong evictions is 16, and transfer interval is every 16th page fault. For the following performance evaluation, we use these values as default parameters.

B. Performance

We compare HPE against two well-known policies, LRU and random, as well as two advanced replacement policies, CLOCK-Pro and RRIP. For HPE, we turned on dynamic adjustment and used HIR to record page walk hit information. The transfer latency was calculated and added to the execution time. To simplify simulation, for LRU, CLOCK-Pro, and RRIP, we used an ideal model in which both page walk hits and page faults update their chains (in the exact reference order) directly without any transfer latency. We use IPC and number of evictions for comparison.

Compared to LRU: Fig. 10 compares the performance of HPE and LRU. We make four observations. First, despite considering transfer latency, HPE still outperforms LRU on average. Second, for LRU-friendly applications (types I and VI), HPE performs similarly to LRU. Third, for applications of types III–V, HPE achieves slightly better average performance than LRU. Fourth, for LRU-averse applications (type II), HPE achieves much better performance, with the highest speedup up to 2.81× (*HSD*). On average, HPE achieves 1.34× and 1.16× speedup over LRU when the oversubscription rate is 75% and 50%, respectively.

HPE performs slightly worse than LRU for several applications, such as *NW*, *SAD*, *MVT*, and *HWL*. *NW* touches even and odd pages on different occasions. HPE addresses this issue by dividing a page set into two parts; however, some page sets do not meet the division requirement. We find if more page sets are divided by relaxing the division requirement, the performance of *NW* can be improved (result is not shown). In most cases, *MVT* touches pages with an address stride of 4. This causes way-conflict in HIR due to wasted entry space (only records four pages in a page set). As a result, some page walk hit information is lost. For *SAD*, HPE incurs instant thrashing. For *HWL*, HPE evicts slightly more pages due to loss of an exact reference order.

Fig. 11 compares evictions of HPE and LRU. We make three observations. First, for LRU-friendly applications (types I and VI), HPE evicts a similar quantity of pages as LRU. Second, for applications of types III–V, HPE evicts slightly fewer pages on average. Third, for LRU-averse applications, HPE evicts much fewer pages. On average, HPE evicts 18% and 12% fewer pages than LRU when the oversubscription rate is 75% and 50%, respectively.

Compared to Other Policies: We also compare HPE with random, RRIP, and CLOCK-Pro. For RRIP, we use the frequency priority (FP) policy. To alleviate instant thrashing, we enhance RRIP by adding a delay field, which records the global page fault number. Correspondingly, we modify RRIP's

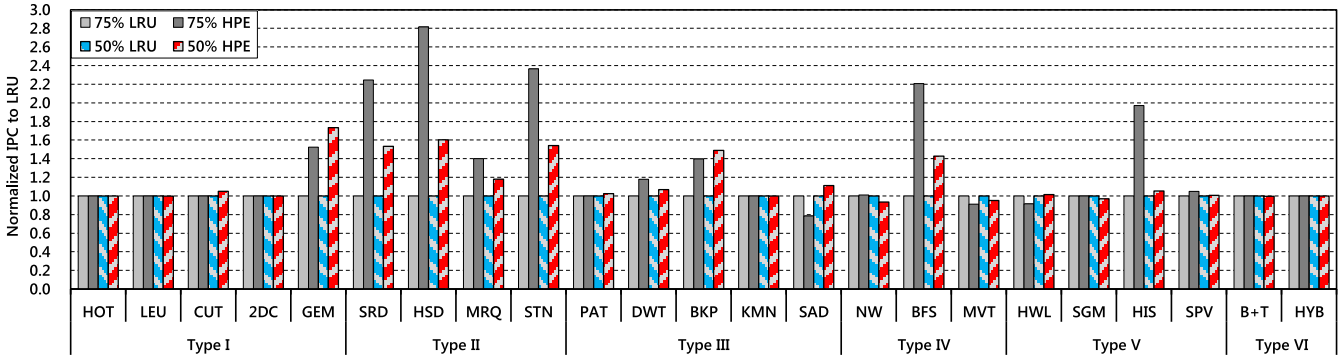


Fig. 10. Comparing HPE's performance to LRU.

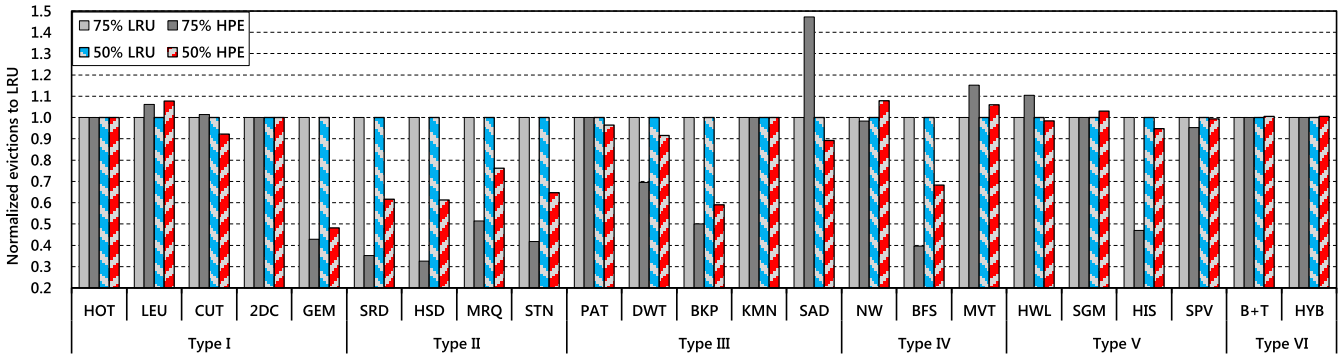


Fig. 11. Comparing HPE's evictions to LRU.

requirement to select an eviction candidate as: 1) a page has distant re-reference interval prediction (same as original RRIP) and 2) the margin between current page fault number and a page's delay field is large than or equal to a threshold (added requirement). For applications with type II, RRIP inserts pages with distant re-reference interval prediction, and the threshold is set to 128. For applications of other access patterns, RRIP inserts pages with long re-reference interval prediction, and the threshold is 0. For CLOCK-Pro, we set m_c (memory allocations for cold pages) [16] to 128 because this value can alleviate instant thrashing. The result is normalized to the ideal policy, which is shown in Fig. 12.

We make three observations. First, on average, HPE performs better (evicts fewer pages) than random, RRIP, and CLOCK-Pro, especially for applications of types II and VI. Second, random is competitive with LRU for most access patterns except typed IV and VI, which corroborates prior observation [10]. Third, random, RRIP, and CLOCK-Pro perform worse than LRU for type VI. HPE addresses this issue by selecting eviction candidates from the LRU position of the old partition. On average, when the oversubscription rate is 75%, HPE is within 11% of Ideal, and evicts 18% more pages than Ideal. It achieves 1.16×, 1.27×, and 1.2× speedup over random, RRIP, and CLOCK-Pro, respectively. When the GPU memory is 50% oversubscribed, HPE is within 11% of Ideal and evicts 16% more pages than Ideal. It achieves 1.21×, 1.16×, and 1.15× speedup over random, RRIP, and CLOCK-Pro, respectively.

Sensitivity to Page Walk Latency: To study how page walk latency impacts overall performance, we evaluated LRU and HPE with a larger page walk latency of 20 cycles. This result is not shown due to space limitations. We found that LRU and HPE have minimal performance difference between 8 and 20 cycles. Generally, for these values, page walk latency has little influence on overall performance, and this variation in page walk latency has minimal effect on eviction decisions.

C. Overhead Analysis

Search Overhead: MRU-C searches from the MRU position of the old partition until a qualified page set is found, which introduces search overhead. To quantitatively evaluate this overhead, we calculated the average number of comparisons for all searches. Applications that use LRU for the entire execution were omitted. For applications that trigger dynamic adjustment, we only counted the search overhead of MRU-C.

We first studied how often dynamic adjustment was triggered for each application under oversubscription rates of 75% and 50%. We find that *KMN*, *NW*, *B+T*, *HYB*, *SPV*, and *MVT* used LRU for the entire execution under both rates. *HOT*, *BKP*, *PAT*, *LEU*, *CUT*, *MRQ*, *STN*, *2DC*, and *GEM* used MRU-C through their entire execution under both rates. *DWT*, *HSD*, and *SGM* used MRU-C for the entire execution under oversubscription rate of 75%, and switched strategy under oversubscription rate of 50%. The rest of the applications (*SRD*, *BFS*, *SAD*, and *HIS*) switched strategy under both rates.

The details are shown in Fig. 13. In this figure, the percentage after an application on the y-axis is the oversubscription rate. The horizontal line represents normalized percent of time a strategy is used by HPE. It also shows the exact adjustment

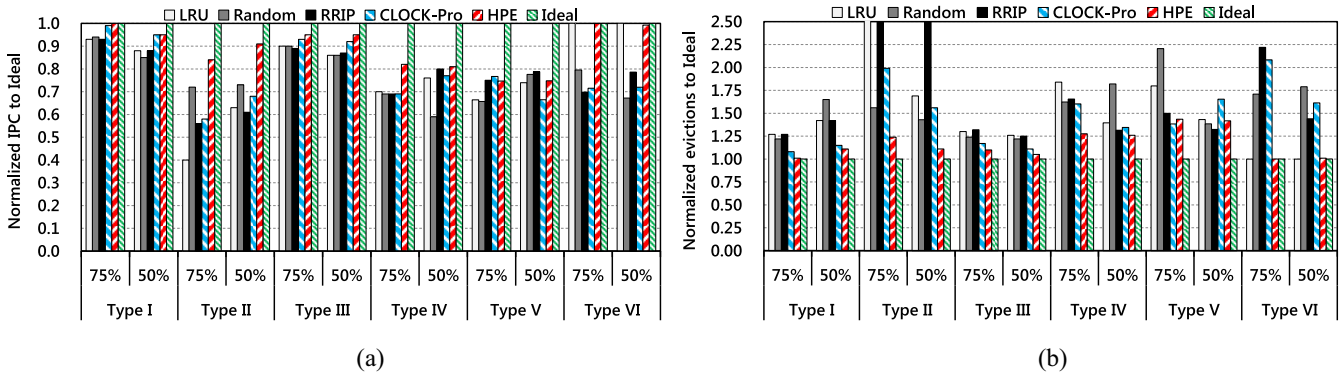


Fig. 12. Comparing HPE's performance and evictions to other policies. (a) Performance comparison. (b) Evictions comparison.

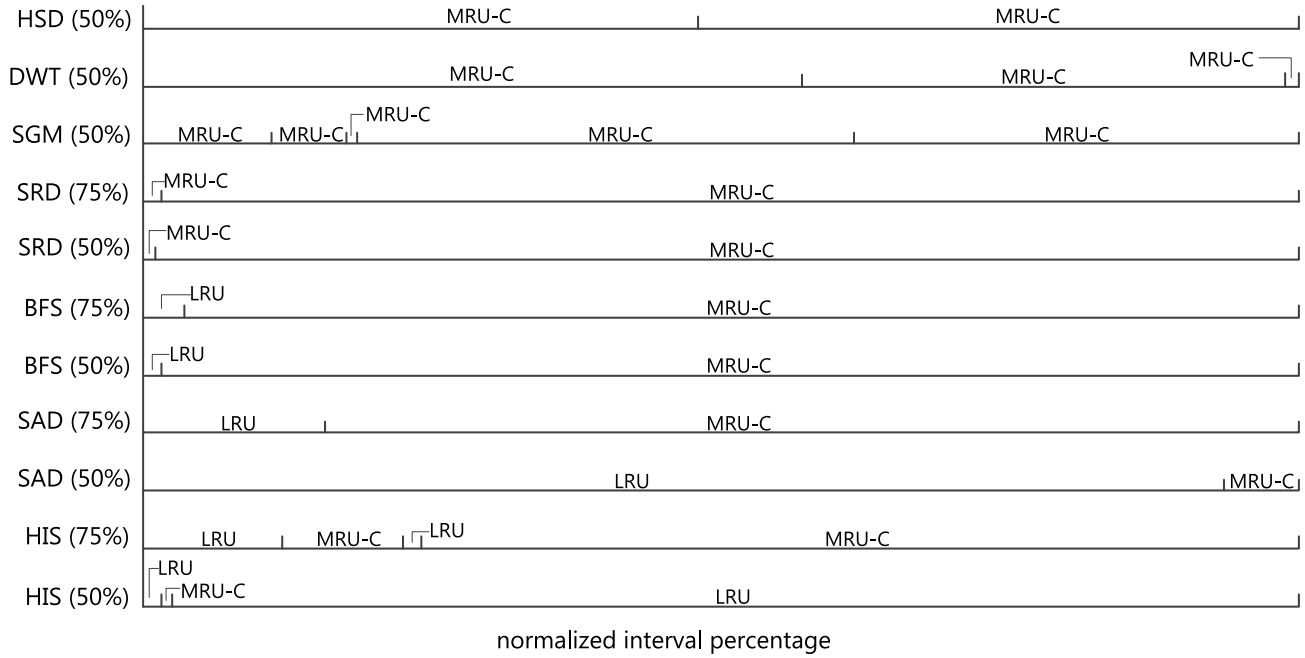


Fig. 13. Break down of eviction strategy adjustment for selected applications.

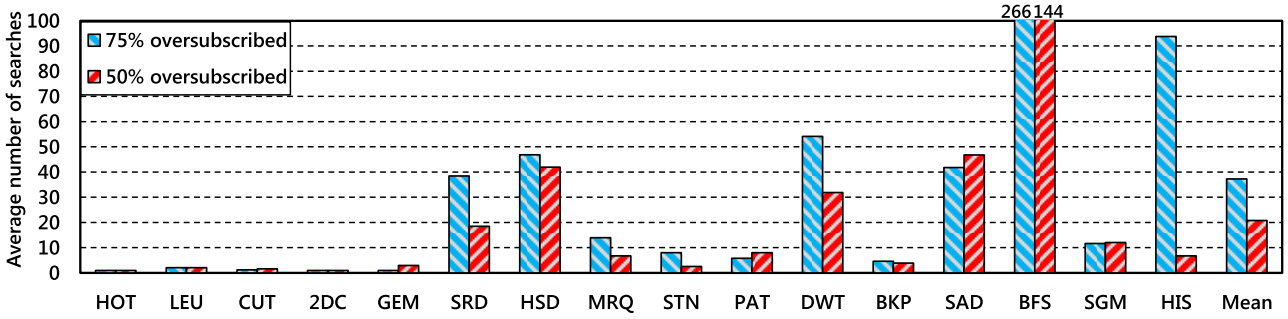


Fig. 14. Average search overhead of HPE.

process. For example, when the GPU memory is 75% oversubscribed, *BFS* first uses LRU for a short duration, and then switches to MRU-C and uses this strategy to application completion. We see that only a small portion of applications trigger dynamic adjustment. *BFS*, *SAD*, and *HIS* switch between LRU and MRU-C; the rest of the applications (*SRD*, *HSD*, *DWT*, and *SGM*) adjust search point.

Fig. 14 shows the search overhead of each application. For most applications, the average number of comparisons is rather small, typically smaller than 50, with outliers of *BFS* and *HIS*. These applications are classified as **irregular#2**, and they experience dynamic adjustment during runtime, which can be seen from Fig. 13. For *BFS*, MRU-C was used for most of its execution. *HIS* used MRU-C (LRU) for most of its execution

under oversubscription rate of 75% (50%). We measured wall clock time to conduct 300 comparisons in a list, and we found that the comparison time is a small portion (19.92% on average) of the page fault penalty (20 μs). For applications whose average number of comparisons is lower than 50, the search time is negligible.

Classification Overhead: HPE relies on statistics to classify applications into three categories. The classification does not analyze the access patterns (Fig. 2) in depth. Instead, it just needs to traverse the page set chain sequentially, classify page set counters into three categories, and finally compare the result with threshold values. To quantitatively measure this latency, we used the data of *KMN* to evaluate the time for classification, because it has the largest footprint, and hence, it introduces the highest overhead. We included all operations for classification. The average time for classification is 16.7 μs, a bit smaller than the page fault penalty (20 μs). As classification is performed only once during execution, it has negligible impact on overall performance.

Overhead of HIR: We used an 8-way HIR with 1024 entries to record page walk hit information. Each entry has a tag and data field. Assume the system is 64-bit wide, the page size is 4 KB and page set size is 16. Therefore, the tag is 48 bits. The data has 16 counters, and a 2-bit counter can meet the requirement in most cases. Thus, the data field needs 32 bits in total. An entry needs 80 bits (10 B), and 1024 entries incur 10-KB storage cost. As HIR is shared by all SMs, the storage cost is not significant, which only translates to an overhead of 4.2% of all SMs' L1 data cache (240 KB).

Overhead on the CPU Side: Current GPUs rely on a software driver, which runs on the host processor, to handle page faults in a unified memory. HPE requires some extensions to this driver, which introduces some overhead. The overhead includes storage cost for page walk hit information and the time to update the page set chain. As the size of HIR is 10 KB, a 10-KB buffer in the host's main memory is enough to store all transferred information. For modern computer systems with large memory, this storage cost is negligible.

There is also some cost to update the page set chain in the GPU driver. This cost has two parts. First, the time to transfer page walk hit information from the GPU to the GPU driver on the host. Second, the time spent in the GPU driver to update the page chain set. Both the transfer time and the update time depend on the number of populated entries in HIR. As Fig. 15 shows, the number of populated entries is typically small, i.e., less than ten in most applications. *MVT* has the largest value (139). We have considered the transfer latency when evaluating HPE. To approximate the cost for update, we measured the wall clock time to perform an update of 150 records in a hashmap data structure. We set the page set chain length to 200, which is larger than *MVT*'s 180 (average). In this scenario, the average update time is 16.1 us, which is 80% of the page fault penalty (20 μs). This overhead is acceptable for three reasons. First, the update is done every 16th page fault, and the overhead is amortized over 16 page faults. Thus, the overhead is 5% of page fault penalty. Second, the update of the page chain set is not on the critical path of page fault handling, and does not delay the handling of faults. Lastly, this approximation assumes the worst-case based on *MVT*. As Fig. 15 shows, most applications require much fewer updates (less than ten in majority).

Because the GPU driver executes on the host processor, there is load imposed on a host CPU core, regardless of the policy. To estimate HPE's additional overhead, we use core load, i.e., utilization, to represent resource consumption. The load is estimated as core busy time, including handling page faults and updating the page set chain, divided by total application execution time (i.e., total GPU cycles). For simplicity, we use fixed, worst-case values (16.1 μs) for the additional cost to update the page chain set by HPE. For LRU, the average core load is 29.9% and 39.3% for an oversubscription rate of 75% and 50%, respectively. For RRIP, the average core load is similar to LRU, around 30.3% and 39.5%, respectively. For CLOCK-Pro, the average core load is 29.5% and 39.2%, respectively. In comparison, HPE has a slightly higher load of 34.0% and 47.2%, respectively.

HPE's higher load is likely due to the way that we estimated load. In particular, we made several simplifications for LRU, RRIP, and CLOCK-Pro that favored these schemes. First, we used an ideal model for LRU, RRIP, and CLOCK-Pro, where both page walk hits and page faults update the chains in the exact reference order. We did not consider their transfer latency and update overhead (related to page walk hits) when estimating core load. Second, HPE manages its chain at the page set level. LRU, RRIP, and CLOCK-Pro manage their chains at the page level. We did not consider the higher cost of the search required by these policies. Finally, we used the worst-case update cost of 16.1 μs for estimating core load for HPE. Due to these simplifications and fewer page faults with HPE, in practice, we expect HPE's load to be similar, or possibly lower, than LRU, RRIP, and CLOCK-Pro.

VI. Related Work

As this article aims to design a page eviction policy for GPUs with unified memory, in this section, we discuss previous research related to unified memory and eviction policies.

A. Unified Memory

Unified memory requires the support for address translation. Power *et al.* [17] and Pichai *et al.* [20] were among the first to explore necessary components for GPU address translation. Power *et al.* showed that per-SM post-coalescer TLBs, a highly threaded page table walker, and a shared page walk cache are essential components for efficient address translation. Ausavarungnirun *et al.* [11] showed the performance advantage of a shared L2 TLB over a shared page walk cache. Vesely *et al.* [26] analyzed shared virtual memory for on-die GPUs using real-system measurements. They found that serving TLB misses and page faults from the GPU are much slower than serving that from the CPU.

Based on these work, Zheng *et al.* [10] proposed replayable far-faults to enable SMs to continue execution in the presence of page faults. Ausavarungnirun *et al.* proposed *Mosaic* [11], a GPU memory manager that provides application-transparent support for multiple page sizes, which uses base pages to transfer data over PCIe and allows TLBs to use large pages

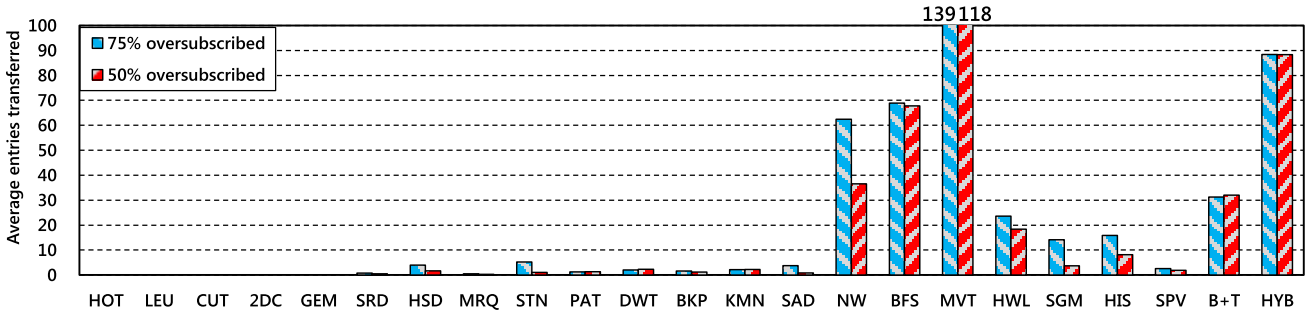


Fig. 15. Average number of entries transferred each time.

to reduce address translation overhead. They also proposed *MASK* [21], a new GPU framework that provides low-overhead virtual memory support for concurrent execution of multiple applications. Shin *et al.* [18] proposed an SIMT-aware page walk scheduler that batches requests from the same instruction to be serviced together and prioritizes walk requests from instructions that would require less work. They also proposed mechanisms to coalesce the address translation needs of all pending page table walks in the same neighborhood that happen to have their address mappings fall in the same cache line [19]. These techniques are orthogonal to this article and we applied the replayable far-faults in our infrastructure.

B. Eviction Policies

A significant amount of research has targeted improving performance of eviction policies for cache and memory on CPUs. The widely used recency-based policy LRU performs well for a significant portion of applications; however, it performs poorly for thrashing access patterns. To address this issue, Qureshi *et al.* proposed DIP [30], which places the incoming cache blocks in the LRU position rather than MRU position. Jaleel *et al.* proposed RRIP [15], a policy that uses an M-bit register (per cache block) to store its re-reference prediction value and evicts a cache block with a distant re-reference interval. Both DIP and RRIP use *set dueling* [31] to be adaptive. However, we find that instant thrashing occurs if they are applied to unified memory. Besides, it is not easy to apply set dueling in memory. LFU is a representative frequency-based policy. We find that using frequency information is not enough to select appropriate eviction candidates in unified memory.

In practice, ideal LRU is too expensive to implement, and an approximation is usually used, such as not recently used (NRU) and the CLOCK algorithm. However, NRU and CLOCK inherit the inefficiency of LRU when dealing with thrashing access patterns. Some researchers proposed variants of CLOCK to address this issue. Carr and Hennessy proposed WSClock [32], which combines a WS algorithm and CLOCK. Bansal and Modha proposed CAR [33], which combines the advantages of adaptive replacement cache (ARC) [34] and CLOCK. Jiang *et al.* proposed CLOCK-Pro [16], which integrates the principle of LIRS [35] and CLOCK.

For unified memory, Zheng *et al.* evaluated the performance of LRU and random for some applications, which demonstrates that random policy is competitive with LRU. However, we find that without page reference information, random may incur significant performance degradation in some cases. This article aims to propose a page eviction policy with high performance and low overhead for GPUs. This article reduces the management overhead by maintaining a chain at the page set level, and relies on classification to select appropriate eviction strategy for each category.

VII. Conclusion

In this article, we proposed a new page eviction policy called HPE. This policy manages a page set chain dynamically and uses statistics to classify applications into three categories. Based on the classification result, HPE selects an appropriate eviction strategy. HPE also applies dynamic adjustment to adapt the eviction strategy when necessary. In addition, we add a very small set-associative cache on the GPU side to record page walk hit information. This information is transferred to the GPU driver periodically to update the page set chain. The simulation results show that, on average, HPE achieves 1.34× and 1.16× speedup over LRU when the oversubscription rate is 75% and 50%, respectively. HPE also outperforms RRIP and CLOCK-Pro.

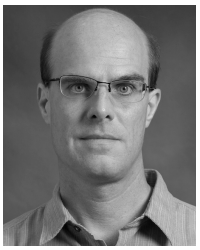
References

[1] NVIDIA. (2016). *GPU-Accelerated Applications*. [Online]. Available: http://images.nvidia.com/content/tesla/pdf/Apps-Catalog-March-2016.pdf
[2] N. Agarwal, D. W. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler, "Selective GPU caches to eliminate CPU-GPU HW cache coherence," in *Proc. 22nd IEEE Int. Symp. High Perform. Comput. Architect.*, 2016, pp. 494–506.
[3] A. Arunkumar *et al.*, "MCM-GPU: Multi-chip-module GPUs for continued performance scalability," in *Proc. 44th Annu. Int. Symp. Comput. Architect.*, 2017, pp. 320–332.
[4] *CUDA C Programming Guide*, NVIDIA, Santa Clara, CA, USA, 2011.
[5] *OpenCL—The Open Standard for Parallel Programming of Heterogeneous Systems*, Khronos OpenCL Working Group, Beaverton, OR, USA, 2011.
[6] J. Luitjens. (2014). *CUDA Streams: Best Practices and Common Pitfalls*. [Online]. Available: http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf
[7] M. Harris. (2013). *Unified Memory in CUDA 6*. [Online]. Available: https://devblogs.nvidia.com/unified-memory-in-cuda-6/
[8] G. Kyriazis, "Heterogeneous system architecture: A technical review," Santa Clara, CA, USA, AMD, White Paper, 2012.
[9] N. Sakharnykh. (2016). *Beyond GPU Memory Limits With Unified Memory on Pascal*. [Online]. Available: https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/

[10] T. Zheng, D. W. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards high performance paged memory for GPUs," in *Proc. 22nd IEEE Int. Symp. High Perform. Comput. Architect.*, 2016, pp. 345–357.
[11] R. Ausavarungnirun *et al.*, "Mosaic: A GPU memory manager with application-transparent support for multiple page sizes," in *Proc. 50th IEEE/ACM Int. Symp. Microarchitect.*, 2017, pp. 136–150.
[12] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.
[13] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Center Reliable High Perform. Comput., Univ. Illinois at Urbana–Champaign, Champaign, IL, USA, Rep. IMPACT-12-01, 2012.
[14] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Proc. Innov. Parallel Comput.*, 2012, pp. 1–10.
[15] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. 37th Int. Symp. Comput. Architect.*, 2010, pp. 60–71.
[16] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An effective improvement of the CLOCK replacement," in *Proc. USENIX Annu. Tech. Conf.*, 2005, pp. 323–336.
[17] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *Proc. 20th IEEE Int. Symp. High Perform. Comput. Architect.*, 2014, pp. 568–578.
[18] S. Shin *et al.*, "Scheduling page table walks for irregular GPU applications," in *Proc. 45th Int. Symp. Comput. Architect.*, 2018, pp. 180–192.
[19] S. Shin, M. LeBeane, Y. Solihin, and A. Basu, "Neighborhood-aware address translation for irregular GPU applications," in *Proc. 51st IEEE/ACM Int. Symp. Microarchitect.*, 2018, pp. 352–363.
[20] 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 *Proc. 19th ACM Int. Conf. Architect. Support Program. Lang. Oper. Syst.*, 2014, pp. 743–758.
[21] R. Ausavarungnirun *et al.*, "MASK: Redesigning the GPU memory hierarchy to support multi-application concurrency," in *Proc. 23rd ACM Int. Conf. Architect. Support Program. Lang. Oper. Syst.*, 2018, pp. 503–518.
[22] J. Kehne, J. Metter, and F. Bellosa, "GPUswap: Enabling oversubscription of GPU memory through transparent swapping," in *Proc. 11th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2015, pp. 65–77.
[23] N. Agarwal, D. W. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking bandwidth for GPUs in CC-NUMA systems," in *Proc. 21st IEEE Int. Symp. High Perform. Comput. Architect.*, 2015, pp. 354–365.
[24] N. Sakharnykh. (2017). *Unified Memory on Pascal and Volta*. [Online]. Available: http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf
[25] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2009, pp. 163–174.
[26] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee, "Observations and opportunities in architecting shared virtual memory for heterogeneous systems," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2016, pp. 161–171.
[27] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.
[28] NVIDIA. (2016). *NVIDIA Tesla P100*. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
[29] J. Hestness, S. W. Keckler, and D. A. Wood, "A comparative analysis of microarchitecture effects on CPU and GPU memory system behavior," in *Proc. IEEE Int. Symp. Workload Characterization*, 2014, pp. 150–160.
[30] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. S. Emer, "Adaptive insertion policies for high performance caching," in *Proc. 34th Int. Symp. Comput. Architect.*, 2007, pp. 381–391.
[31] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for MLP-aware cache replacement," in *Proc. ISCA*, 2006, pp. 167–178.
[32] R. W. Carr and J. L. Hennessy, "WSCLOCK—A simple and effective algorithm for virtual memory management," *ACM SIGOPS Oper. Syst. Rev.*, vol. 15, no. 5, pp. 87–95, 1981.
[33] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," in *Proc. 3rd USENIX Conf. File Storage Technol.*, 2004, pp. 187–200.
[34] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. 3rd USENIX Conf. File Storage Technol.*, 2003, pp. 115–130.
[35] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 1, pp. 31–42, 2002.

Qi Yu received the B.S. degree in computer science and technology from Xi'an Jiaotong University, Xi'an, China, in 2013, and the M.S. degree in computer science and technology from the National University of Defense Technology, Changsha, China, in 2015, where he is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology.

His current research interests include GPU architecture and memory access pattern analysis.



Bruce Childers (M'00) received the B.S. degree in computer science from the College of William and Mary, Williamsburg, VA, USA, in 1991, and the Ph.D. degree in computer science from the University of Virginia, Charlottesville, VA, USA, in 2000.

He is a Professor with the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA, USA. His current research interests include intersection of the software-hardware boundary for improved energy, performance, reliability in computer systems design, with an emphasis on embedded systems, technology and cultural changes to advance transparency, reuse, and reproducibility in computationally driven science.

Prof. Childers is a member of the IEEE Computer Society.

Libo Huang received the B.S. and Ph.D. degrees in computer engineering from the National University of Defense Technology, Changsha, China, in 2005 and 2010, respectively.

He was an Associate Professor with the Department of Computer Science, National University of Defense Technology in 2014. He has authored over 60 papers in internationally recognized journals and conferences. His current research interests include computer architecture, hardware/software codesign, VLSI design, and on-chip communication.

Cheng Qian received the B.S., M.S., and Ph.D. degrees in computer science and technology from the National University of Defense Technology, Changsha, China, in 2012, 2014, and 2018, respectively.

His current research interests include 3-D memory architecture design and memory access scheduling mechanism.

Zhiying Wang received the Ph.D. degree in electrical engineering from the National University of Defense Technology (NUDT), Changsha, China, in 1988.

He is currently a Professor with the School of Computer, NUDT. He has contributed over 10 invited chapters to book volumes, published 240 papers in archival journals and refereed conference proceedings, and delivered over 30 keynotes. His current research interests include computer architecture, computer security, VLSI design, reliable architecture, multicore memory system, and asynchronous circuit.

Prof. Wang is a member of CCF and ACM.