Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory

Debashis Ganguly
Department of Computer Science
University of Pittsburgh
debashis@cs.pitt.edu

Ziyu Zhang
Department of Computer Science
University of Pittsburgh
ziz41@pitt.edu

Jun Yang
Electrical and Computer Engineering Department
University of Pittsburgh
juy9@pitt.edu

Rami Melhem
Department of Computer Science
University of Pittsburgh
melhem@cs.pitt.edu

ABSTRACT

Memory capacity in GPGPUs is a major challenge for data-intensive applications with their ever increasing memory requirement. To fit a workload into the limited GPU memory space, a programmer needs to manually divide the workload by tiling the working set and perform user-level data migration. To relieve the programmer from this burden, Unified Virtual Memory (UVM) was developed to support on-demand paging and migration, transparent to the user. It further takes care of the memory over-subscription issue by automatically performing page replacement in an oversubscribed GPU memory situation. However, we found that naïve handling of page faults can cause orders of magnitude slowdown in performance. Moreover, we observed that although prefetching of data from CPU to GPU can hide the page fault latency, the difference among various prefetching mechanisms can lead to drastically different performance results. To this end, we performed extensive experiments on `GeForceGTX 1080ti` GPUs with PCI-e 3.0 16x to discover that there exists an effective prefetch mechanism to enhance locality in GPU memory. However, as the GPU memory is filled to its capacity, such prefetching mechanism quickly proves to be counterproductive due to locality unaware eviction policy. This necessitates the design of new eviction policies that are aware of the hardware prefetcher semantics. We propose two new programmer-agnostic, locality-aware pre-eviction policies which leverage the mechanics of existing hardware prefetcher and thus incur no additional implementation and performance overhead. We demonstrate that combining the proposed tree-based pre-eviction policy with the hardware prefetcher provides an average of 93% and 18.5% performance speed-up compared to LRU based $4KB$ and $2MB$ page replacement strategies, respectively. We further examine the memory access pattern of GPU workloads under consideration to analyze the achieved performance speed-up.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322224

CCS CONCEPTS

• **Computer systems organization** → **Single instruction, multiple data**; *Heterogeneous (hybrid) systems*; • **Computing methodologies** → *Graphics processors.*

KEYWORDS

unified virtual memory, GPU, hardware prefetcher, page eviction policy

ACM Reference Format:
Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307650.3322224

1 INTRODUCTION

Graphics Processing Units (GPUs) rely on massive thread-level parallelism (TLP) to hide memory system latency. Steady increase in GPU compute density and advancement in high-level programming tools further led to a wider adoption of GPUs by general purpose applications in high-performance computing. However, the major roadblock in the general adaptation of GPUs is the traditional "copy then execute" programming model as the onus of maintaining complex data structure, tiled data transfer, explicit data migration falls on the application developers. To address this, AMD [2] and NVIDIA [19] have released new discrete GPU architectures with runtime support for Unified Virtual Memory (UVM). UVM allows both CPU and GPU to reference data using a shared pointer to the unified virtual address space. Memory is allocated and migrated on-demand. Further, older pages in GPU memory are replaced transparently to make room for new migrations under strict memory budget.

With demand paging, GPUs face a new type of page-faults (called far-faults henceforth) which occur when the data is not present in the device memory. These far-faults are resolved by the software runtime resident to the host processor and contribute to significant performance overhead. Page granularity in current NVIDIA GPUs is $4KB$. Stalling kernel execution for every $4KB$ page migration over PCI-e interconnect is not a viable option. Prefetching memory in advance to increase L2 cache hit rate has been extensively studied [11–13] for GPUs. Whereas, prefetching is little explored for

GPU physical memory in the context of UVM. Zheng et al [26] first proposed both user-directed prefetchers and hardware prefetchers in the context of GPU's unified memory management. That work shows that instead of piecemeal migration of pages on-demand, prefetching larger chunks of memory improves PCI-e utilization and reduces transfer latency. Further, prefetched pages reduce the number of far-faults and in turn the overhead to resolve them. CUDA 8.0 runtime [18] also implements their own proprietary hardware prefetcher. We created a set of micro-benchmarks to uncover the exact mechanics of the locality-aware tree-based neighborhood prefetcher implemented in NVIDIA `GeForce GTX 1080ti` GPUs. We show that compared to other common prefetchers and on-demand paging this tree-based hardware prefetcher provides a huge performance speed up. This prefetcher can migrate multiples of 64*KB* basic blocks contiguous in the virtual address space grouped in a single transfer. All pages being prefetched are local to the current faulty pages and are within 2*MB* large page boundary. Prefetching pages within 2*MB* neighborhood is to leverage the support for large pages by the page table in modern 64-bit processors.

However, under strict memory budget, aggressive prefetching can be counterproductive displacing other heavily referenced pages from memory. A simple solution is to disable hardware prefetcher under memory over-subscription. When the GPU memory usage has reached its capacity and the workload still needs on-demand page migration to continue further execution, GPU must identify page(s) for eviction. Least Recently Used (LRU) based 4*KB* page eviction does not help the cause of hardware prefetcher. As pages in the LRU list can be far spaced in the virtual address space, it breaks the semantics of locality aware prefetcher. Moreover, under over-subscription, resolving far-faults becomes costlier as new migration stalls for writing back older pages. Researchers have also explored memory-threshold based pre-eviction policy that maintains a constant free-page buffer from where memory can be allocated directly without waiting for writing back dirty pages. However, in the presence of hardware prefetcher, maintaining a constant pool of free pages becomes challenging. To work around 4*KB* page eviction, memory eviction granularity in NVIDIA GPUs is 2*MB*. Like aggressive prefetching, aggressive eviction has adverse effect on performance. 2*MB* memory eviction can cause a large page thrashing for repetitive kernel launches. This necessitates careful investigation and design of new locality-aware page replacement strategies compatible with locality-aware hardware prefetcher.

To the best of our knowledge, this is the first work that profiles the semantics of the hardware prefetcher supported by NVIDIA GPUs and then further analyzes the interplay between hardware prefetcher and page eviction policy in CPU-GPU unified memory. We propose two new pre-eviction policies inspired by the hardware prefetcher. These pre-eviction schemes are aware of the hardware prefetcher semantics. They pre-evict contiguous pages in multiples of 64*KB* basic block around an eviction candidate chosen from the LRU page list. Pre-evicting contiguous pages in bulk the way they were brought in by the prefetcher allows further prefetching under memory constraint. Moreover, the proposed tree-based neighborhood pre-eviction scheme is adaptive where the eviction size varies between the two extremities of static eviction granularity: 4*KB* and 2*MB*. We show that combining these new pre-eviction policies with prefetcher provides almost an order of magnitude performance speedup compared to LRU based 4*KB* page replacement. We further analyze the memory access pattern of the GPU workloads to get more insight into the achieved performance speed-up.

This paper makes the following contributions:

(1) We identify the semantics of tree-based neighborhood prefetcher implemented by NVIDIA CUDA drivers using a set of detailed experiments. We reinforce the necessity of such hardware prefetcher in the success of UVM by comparing against on-demand paging and other proposed hardware prefetchers.
(2) We demonstrate how LRU 4*KB* eviction policy breaks the semantics of a locality-aware prefetcher and leads to a rapid performance degradation under memory over-subscription. We also show that in contrary to the popular belief, traditional memory threshold-based pre-eviction policies have adverse effects in the presence of a hardware prefetcher.
(3) We propose two new locality aware pre-eviction policies: sequential-local and tree-based neighborhood scheme that are inspired by the hardware prefetcher and respect its semantics.
(4) We show that combining these pre-eviction policies with hardware prefetcher provides dramatic performance improvement under over-subscription.
(5) After carefully examining memory access patterns of the used benchmarks, we follow a simple optimization of reserving a percentage of pages from eviction from the top of LRU list and show that this ensures better performance for workloads with data reuse over multiple kernel launches.

2 BACKGROUND

In this section, we describe the GPU execution model and the baseline architecture. Our description closely follows NVIDIA/CUDA terminology in specific cases, however, it is general enough to describe any vendor agnostic discrete CPU-GPU system. We further discuss about the CPU-GPU Unified Virtual Memory (UVM).

2.1 GPU Execution Model and Architecture

Despite extensive academic researches and industrial investments in on-die integrated GPUs, discrete GPUs combined with CPUs dominate the heterogeneous computing spectrum. GPUs are connected to the host CPU system through the PCI-e interconnect. A GPU program consists of two parts: host code and device code or GPU kernel. GPU kernel, written for one thread, is executed by multiple threads on GPU. On GPU, threads are grouped into thread blocks (TBs). The number of TBs and the number of threads per TB is specified by the programmer. These TBs are executed by Streaming Multiprocessors (SMs) which is the main computation unit (CU) of a GPU. A GPU has multiple SMs and each SM consists of a set of simple cores. All SMs share GDDR5 as the device memory through an interconnect network. GPUs also have a unified L2 data cache for all SMs. Memory accesses generated by the SMs are first coalesced by the load/store unit before relaying them to the GPU memory controller. This reduces processing of redundant memory accesses per cycle.

2.2 Unified Memory and On-demand Page Migration

In the classic "copy then execute" model, GPU programmers have to allocate memory on both host and device. Before launching the kernel to the GPU, programmers have to explicitly copy the data from the host to the device and upon completion copy the data back to the host from the device. This model of execution comes with two major challenges. The first is that data migration and kernel execution are serialized and thus the total runtime is the summation of the two. Complicated asynchronous user-directed constructs to overlap data migration and kernel execution are used to address this issue. The second challenge is memory over-subscription. When the working set of the GPU kernel cannot fit in the device memory, the programmers have to painstakingly redefine the data structures and tile the data to transfer back and forth in chunks. To address these two major challenges, NVIDIA [19] and AMD [2] have introduced software runtime to provide the illusion of CPU-GPU Unified Virtual Memory (UVM). Unified Memory provides a single virtual address space accessible from any processor in the system. In CUDA 8.0 [18], `cudaMallocManaged` allows applications to allocate data that can be read or written from code running on either CPUs or GPUs using a single shared pointer.

In the "copy then execute" model, data is always physically available in the device memory before the kernel starts executing. A near-fault can occur upon L2 cache miss. Whereas, with UVM, a new type of page fault is introduced which we will refer to as a *far-fault* henceforth. Upon allocating data using `cudaMallocManaged`, no physical memory space is allocated on either host or device. Rather, on each access, each processor encounters with a far-fault and the memory is allocated and migrated on-demand. As the memory is allocated on-demand, new page table entries (PTEs) are created in the GPU's page table and upon completion of migration, these entries are validated (the valid flags corresponding to these PTEs are set in the page table).

A far-fault is much costlier than a near-fault in terms of the time to resolve as it includes two additional major overheads: a far-fault handling latency (typically 45μs in Pascal GPUs) and data migration latency over PCI-e interconnect. Figure 1 shows a simplified control flow demonstrating how the GPU Memory Management Unit (GMMU) handles a far-fault. This is described as the following sequence of actions performed within a GPU. This model is inspired by the replayable far-faults proposed by Zheng et al [26].

❶ Scheduled threads generate global memory accesses. ❷ Each SM has its own load/store unit. Every load/store unit has its own TLB. Load/store unit performs a TLB look up to find whether the translation for the issued memory access is cached in TLB or not. A TLB miss is relayed to the GMMU. ❸ The GMMU walks through the page table looking for a PTE corresponding to the requested page with valid flag set. A far-fault occurs if there is no PTE for the requested page or the valid flag is not set. Then the far-fault is registered in the Far-fault Miss Status Handling Registers (MSHRs). ❹ The page is scheduled for transfer over CPU-GPU PCI-e interconnect. ❺ A 4KB page is allocated on demand and data is migrated from host to device memory. ❻ The MSHRs are consulted to notify the corresponding load/store unit and the memory access is replayed. A new PTE entry is added to the page table with valid flag set or the valid flag of the existing PTE is set. In addition, a new entry is added to the TLB.

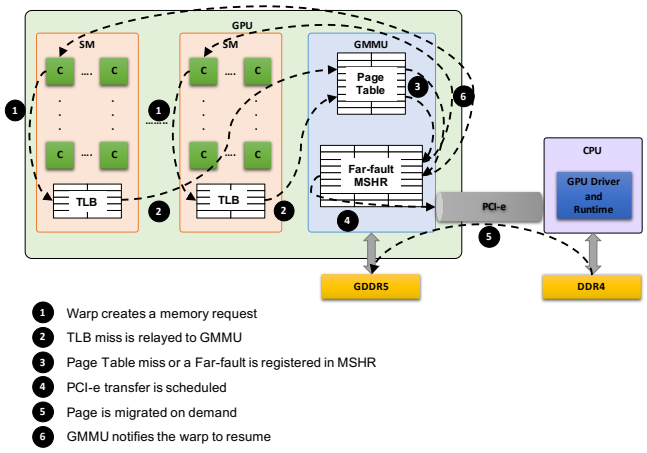


Figure 1: Simplified control flow for far-fault and on-demand page migration in CPU-GPU Unified Memory.

3 HARDWARE PREFETCHERS

With UVM, kernel execution stalls at every far-fault for page allocation and data migration from host to device. The total kernel execution time increases dramatically as it includes far-fault handling latency and memory copy time. `cudaMemPrefetchAsync`, is an asynchronous construct in CUDA 8.0, that allows programmers to specify an address range to migrate in parallel to the kernel execution. Prefetching later referenced pages helps reduce the number of page faults and also ensures overlap between data migration and kernel execution. However, the responsibility of what to prefetch and when to prefetch still belongs to the programmer. Zheng et al [26] are the first to propose programmer-agnostic hardware prefetchers to overlap kernel execution and data migration. They introduced (i) random, (ii) sequential, and (iii) locality-aware hardware prefetchers. Hardware prefetchers take away the burden from the programmer by automatically deciding what and when to prefetch. Following their lead, we have incorporated the following hardware prefetchers in our simulation framework described in Section 6.1.

3.1 Random (R_p) Prefetcher

A random prefetcher prefetches a random 4KB page along with the 4KB page for which the far-fault occurred in the current cycle. The prefetch candidate is selected randomly from the 2MB large page boundary to which the faulty page belongs. This not only helps CUDA workloads with random access pattern, but also selecting from 2MB large page boundary instead of the whole virtual address space helps in cases of locality of memory accesses.

3.2 Sequential-local (SL_p) Prefetcher

Zheng et al [26] describe their sequential prefetcher as the process of bringing a sequence of 4KB pages from the lowest to the highest order of virtual address irrespective of page access pattern

or far-faults. Their locality aware prefetcher migrates consecutive 128 4KB pages (or total 512KB memory chunk) starting from the faulty-page. We propose a different variation called sequential-local hardware prefetcher. Each `cudaMallocManaged` allocation is logically split into multiple 64KB basic blocks. GMMU upon discovering the pages corresponding to the coalesced memory requests are invalid in the GPU page table, first calculates the base addresses of the 64KB logical chunks to which these faulty 4KB pages belong. Thus, GMMU identifies these 64KB basic blocks as prefetch candidates. Further, it divides these candidate basic blocks into prefetch groups and page fault groups based on the position of the faulty page in the current basic block and then schedules them for sequential transfers by the PCI-e interconnect. Prefetching 64KB basic blocks ensures contiguous 16 4KB pages local to the current faulty pages. Note that this is different from the locality-aware prefetcher proposed in [26]. The position of a faulty page can be anywhere within the corresponding 64KB basic block. Further, multiple faulty pages are taken in consideration while choosing a basic block for prefetching and can be grouped within the same 64KB boundary. Although, 512KB prefetch granularity may yield better performance compared to 64KB sequential local, we put forth our proposed version as it requires no additional coordination across multiple 2MB large pages.

3.3 Tree-based Neighborhood (TBN_p) Prefetcher

GPU Technology Conference 2018 [24] briefly mentioned a tree-based hardware prefetcher implemented by NVIDIA CUDA 8.0 driver. Knowledge of the exact semantics of this prefetcher is proprietary to NVIDIA and was never made public. To discover the exact semantics of this hardware prefetcher, we ran a series of micro-benchmarks on `GeForceGTX 1080ti` and profiled the memory accesses using `nvprof`. We name it as tree-based neighborhood prefetcher. We have published these micro-benchmarks for verification [8].

The semantics of TBN_p demands that every `cudaMallocManaged` allocation is first logically divided into 2MB large pages. Then, these 2MB large pages are further divided into logical 64KB basic blocks to create a full binary tree (or a proper binary tree or a 2 tree) per large page boundary. By the definition of a full binary tree, every node has exactly 2 children nodes. The root node of each binary tree corresponds to the virtual address of a 2MB large page and the leaf-level nodes correspond to the virtual addresses of the 64KB basic blocks. If the user-specified size of an allocation is not a perfect multiple of 2MB, then the remainder size of the allocation breaks the principle of a full binary tree. To address this, the remainder allocation is rounded up to the next $2^i * 64KB$ and another full binary tree is created. For example, if the programmer specifies 4MB and 192KB size for a `cudaMallocManaged` allocation, at the time of allocation, GMMU rounds this size up to 4MB and 256KB. Then two full binary trees for 2MB large pages and one full tree for 256KB are created and maintained by the GMMU transparent to the programmer's knowledge. This behavior can also be verified by running the micro-benchmarks we have published.

The maximum memory capacity of a node in the full binary tree can be calculated as $2^h * 64KB$, where h is the height of a node and $h = 0$ at the leaf level. On every far-fault, the GMMU first identifies the 64KB basic block corresponding to the faulty page being requested. With the understanding that upon migrating, 16 pages in the basic block will be validated in the GPU page table, GMMU then recalculates the to-be valid size of its parent and grandparent up to the root node of the tree. Here and henceforth, by valid size we mean the size of all valid pages corresponding to the leaf-nodes belonging to a given node. At any point, if GMMU discovers the to-be valid size of a node is strictly greater than 50% of the maximum memory capacity at this level, it tries to balance the valid sizes between the two children of that node. This balancing process is recursively pushed down to the children which have not reached the maximum valid size quota. This balancing act identifies basic blocks for prefetching. This process continues till no more basic blocks at leaf level can be identified as prefetch candidates and the to-be valid size of any non-leaf node including root is not more than 50% of maximum size capacity at its level.

Prefetching contiguous pages within 2MB boundary tries to ensure allocation of larger contiguous memory and can also help bypass traversing the nested page tables. This helps reduce the time to access memory. For this same reason, in their work [3], researchers introduced the concept of memory defragmentation to swap and coalesce fragmented memory chunks to ensure contiguous physical memory worth of 2MB large page. However, migrating 4KB pages on-demand and then defragmenting the memory space in the runtime has a substantial overhead. Whereas, TBN_p is an adaptive scheme where the prefetch size can vary from 64KB to 1MB based on the access pattern and opportunity of prefetching. Thus, it can get close to 2MB large page locality without causing any additional performance overhead.

TBN_p can be demonstrated with the help of two examples in Figure 2. Both of these examples explain the semantics on 512KB memory chunk for simplicity. These examples use N_h^i to denote a node in the full binary tree, where h is the height of the node and i is the numeric position of the node in that particular level. We further assume initially all pages in this 512KB allocation are invalid with valid bit not set in the GPU's page table and thus every first access to a page causes a far-fault.

In the first example, for the first four far-faults, GMMU identifies the corresponding basic blocks N_0^1, N_0^3, N_0^5, and N_0^7 for migration. In our example, as the first byte of every basic block is accessed, the basic blocks are split into 4KB page-fault groups and 60KB prefetch groups. All memory transfers are serialized in time. After these first four accesses, each of nodes N_0^1, N_0^3, N_0^5, and N_0^7 has 64KB valid pages. Then, GMMU traverses the full tree to update the valid page size for all the parent nodes and thus each node at $h = 1$ (N_1^0, N_1^1, N_1^2, and N_1^3) has 64KB valid pages. When the fifth access occurs, GMMU discovers that N_1^0 and N_2^0 will have 128KB and 192KB valid pages respectively. For N_2^0, the to-be valid size is greater than 50% of the maximum valid size of 256KB. Hence, the right child N_1^1 is identified for prefetching. This decision is then pushed down to the children. This process identifies the basic block N_0^2 as a prefetch candidate. Further, GMMU discovers that after prefetching N_0^2, N_3^0 will have 320KB of valid pages which is more than 50% of the maximum valid size of 512KB. Then, node N_3^0 pushes prefetch request to the node N_2^1 which in turn pushes it

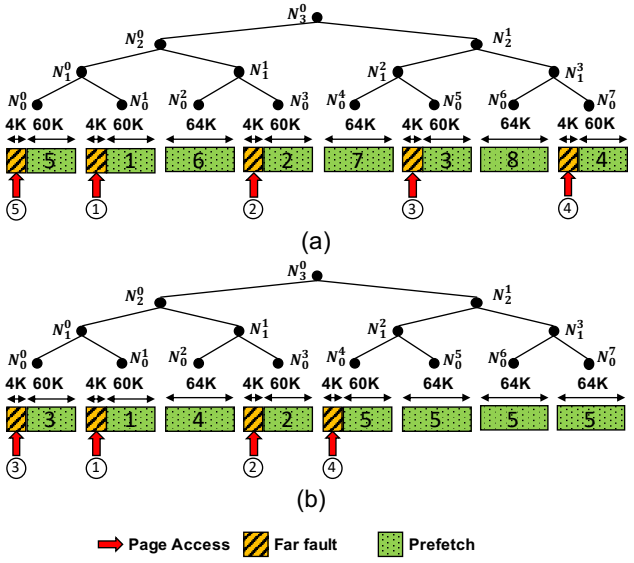


Figure 2: Demonstration of TBN_p on 512 KB memory chunk for two different page access patterns.

to its children. This process identifies basic blocks N_0^4 and N_0^6 for further prefetching.

In the second example, the first two far-faults cause migration of basic blocks N_0^1 and N_0^3. GMMU traverses the tree to update the valid size of nodes N_1^0 and N_1^1 as $64KB$ each. At the third far-fault, as basic block N_0^0 is migrated, the estimated valid sizes for nodes N_1^0, and N_2^0 are updated as $128KB$ and $192KB$ respectively. As the valid size of N_2^0 is more than 50% of the maximum valid size of $256KB$, N_0^2 is identified for prefetching. After this point, the N_2^0 is fully balanced and both N_2^0 and N_3^0 have exactly $256KB$ of valid pages. On fourth access, GMMU discovers that the valid size of N_3^0 will be $320KB$ which is more than 50% of the maximum memory size it can hold. This imbalance causes prefetching of nodes N_0^5, N_0^6, and N_0^7. Note at this point as GMMU finds four consecutive basic blocks, it groups them together to take advantage of higher bandwidth. Then, based on the page fault, it splits this $256KB$ into two transfers: $4KB$ and $252KB$. An interesting point to observe here is that for a full binary tree of $2MB$ size, TBN_p can prefetch at most $1020KB$ at once in a scenario similar to the second example.

4 EFFECTIVENESS OF HARDWARE PREFETCHERS

In this section, we first show the necessity of a hardware prefetcher in CPU-GPU unified memory. Then, we illustrate the criticality of memory over-subscription issue in the presence of a hardware prefetcher. Both the simulation framework and benchmarks used for the experiments in this section are described later in Section 6.1.

4.1 No Over-subscription

In Figure 3, we compare the kernel execution time of the benchmarks using different hardware prefecthing schemes against no hardware prefetching. All hardware prefetchers improve performance significantly compared to just $4KB$ on-demand page migration. This proves the necessity of a hardware prefetcher in CPU-GPU unified memory. The tree-based neighborhood prefetcher provides the best performance compared to the others. This validates the adoption of such scheme in NVIDIA GPU drivers. Note that in this particular experiment, we consider the working sets of the benchmarks to perfectly fit in the available device memory, i.e., there is no over-subscription of memory.

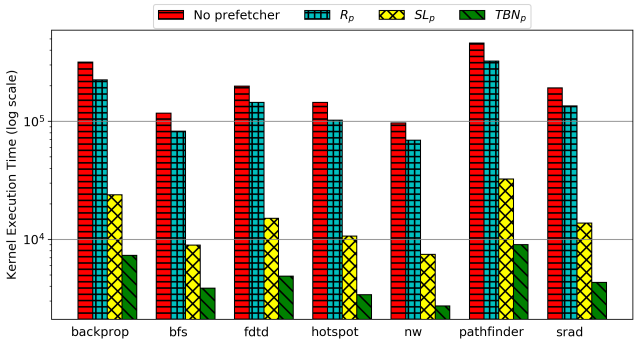


Figure 3: Comparing kernel execution time with different hardware prefetching schemes against no hardware prefetching.

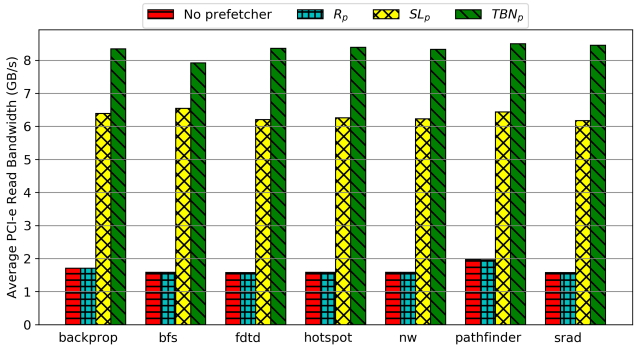


Figure 4: Comparing the average PCI-e read bandwidth for different hardware prefetchers against no hardware prefetching.

To reason behind the performance improvement observed in Figure 3, we plot the average bandwidth of PCI-e read channels in Figure 4. We see that the improvement in kernel performance can be attributed to better PCI-e bandwidth achieved by the corresponding hardware prefetcher.

We ran experiments on `GeForceGTX 1080ti` with `PCI-e 3.0 16x` to find out the maximum attainable PCI-e bandwidth for a given

Table 1: PCI-e read bandwidth measured for different transfer sizes.

Transfer Size (KB)	PCI-e Bandwidth (GB/s)
4	3.2219
16	6.4437
64	8.4771
256	10.508
1024	11.223

transfer size [8]. The findings are enumerated in the Table 1. Irrespective of the transfer size, every PCI-e transaction has a constant activation overhead and cost of setting up the address bus. Thus, scheduling larger transfers amortizes activation overhead and thus reduces transfer latency to guarantee better PCI-e bandwidth. Both on-demand page migration and random prefetcher transfers memory in multiples of $4KB$. Whereas, SL_p can transfer up to $(4+60)KB$ memory chunks and as discussed above TBN_p can migrate maximum $1020KB$ of memory in a single transfer. This is the reason behind the highest PCI-e bandwidth and the best kernel performance achieved by the tree-based neighborhood prefetcher as seen in Figure 4 and 3, respectively.

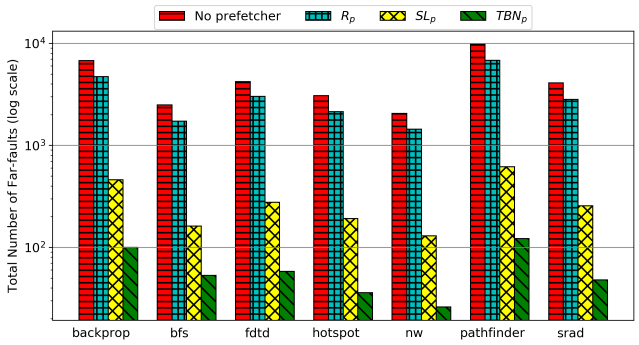


Figure 5: Comparing the total number of far-faults occurred for different hardware prefetchers against no hardware prefetching.

Prefetching pages based on tree-based neighborhood locality reduces the number of far-faults and in turn total far-fault handling latency significantly. Figure 5 shows the total number of far-faults encountered during kernel execution in presence of just on-demand $4KB$ page migration and different hardware prefetchers. Locality-aware prefetching within $2MB$ boundary ensures that prefetched pages are accessed in the immediate future without encountering any far-fault.

4.2 Memory Over-subscription and Pre-eviction

One of the major benefits of Unified Virtual Memory is that the GMMU automatically evicts older pages to make room for the newer page migrations taking care of the over-subscription issue. Aggressive prefetching under memory constraint can be counter-productive as the induced evictions may cause displacement of heavily referenced pages. Thus a natural choice to deal with over-subscription is to disable further prefetching. Further, it is important to select a page for replacement which will not be referenced in the immediate future. LRU and Random **(R_e)** are the two most common page eviction policies [26]. LRU maintains an ordered list of pages based on their last access. Upon reaching GPU memory capacity, LRU chooses the oldest accessed page. Unlike LRU, R_e chooses a random page irrespective of when it is last accessed. GPU Technology Conference 2017 [23] specified that the CUDA drivers implement LRU page replacement policy. In NVIDIA GPUs, the page size is $4KB$. We chose $4KB$ pages as the eviction granularity for this experiment. Moreover, evicting $4KB$ pages based on LRU renders hardware prefetcher ineffective. This is because the SL_p and TBN_p rely on contiguous invalid pages of $64KB$ basic block size which may not be guaranteed after LRU $4KB$ eviction.

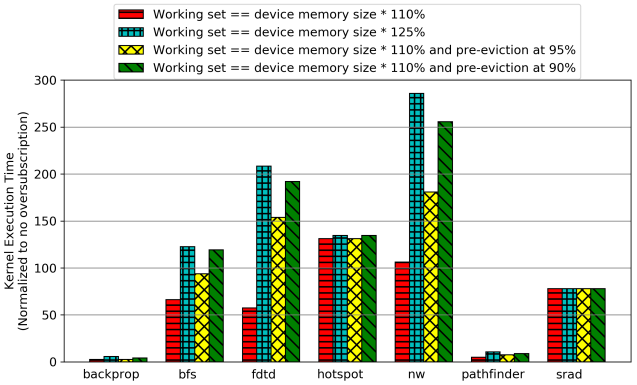


Figure 6: Sensitivity of kernel execution time to the varied percentage of over-subscription and free-page buffer. TBN_p is active before reaching device memory capacity. Upon over-subscription, hardware prefetcher is disabled and pages are migrated at $4KB$ granularity on-demand. LRU $4KB$ is used for eviction.

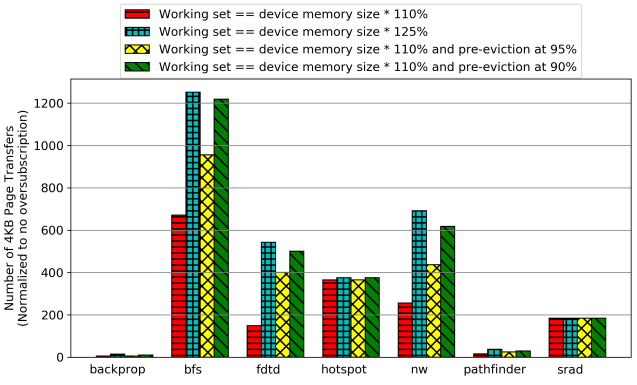


Figure 7: Comparing the total number of 4KB page transfers for varied percentage of over-subscription and free-page buffer.

In this section, we vary the percentage by which the working set is larger than the size of the device memory to demonstrate the sensitivity of kernel execution to the memory over-subscription. Figure 6 shows that in comparison with no over-subscription, the kernel execution degrades drastically even with a small percentage of over-subscription. Note that upon reaching the on-chip memory capacity, the penalty of far-fault is much higher than before. This is because now the threads need to be stalled for writing-back pages along with the latency to migrate new pages. To avoid waiting for write back, in the past, researchers have proposed pre-eviction of pages. A free-page buffer [17] is maintained by causing pre-eviction when the memory occupancy reaches a certain threshold. The kernel execution is not stalled for writing back pages anymore as pages can be allocated from the constant pool of free pages directly. However, Figure 6 shows that it actually hurts the performance. This is because the hardware prefetcher is disabled even before reaching the device memory size capacity to maintain the buffer of free pages.

Upon disabling hardware prefetcher, we lose on the opportunity to benefit from higher interconnection bandwidth as pages are migrated in multiples of 4KB on-demand. Figure 7 shows drastic increase in the number of 4KB page transfers in case of over-subscription and pre-eviction as the hardware prefetcher is disabled when compared against no over-subscription. This explains the performance degradation in Figure 6.

5 PRE-EVICTION POLICIES COMPATIBLE WITH PREFETCHERS

In the previous section, we have showed that a good hardware prefetcher is the key to the success of CPU-GPU Unified Virtual Memory. Both SL_p and TBN_p migrate memory in the multiples of $64KB$ basic block local to the current faulty pages. This is with the hope that the thread blocks will eventually access these pages in the immediate future. However, in reality, some of these pages may not be referenced before eviction procedure starts replacing pages. These unused prefetched pages are never chosen for eviction by LRU. Instead when GPU memory capacity has been reached and kernel execution stalls for new page migration, a heavily referenced page could be chosen for displacement. Thus, an eviction policy, unaware of prefetchers, meets with the challenge how to deal with memory over-subscription issue. A logical choice would be evicting pages in the same way they were brought in by the hardware prefetchers. This means pre-evicting pages in multiples of $64KB$ basic blocks based on sequential or tree-based neighborhood locality. Locality-based pre-eviction has two benefits. Firstly, evicting pages in larger chunks increases PCI-e write-back bandwidth and lowers the write-back latency. Secondly, hardware prefetchers can work in tandem with the pre-eviction scheme. This also means that it overcomes the drawbacks of memory threshold-based pre-eviction policy. To this end, we propose the following two new pre-eviction schemes which we have incorporated in our simulation platform.

5.1 Sequential-local (SL_e) Pre-eviction

Sequential-local eviction consults the LRU page list to select an eviction candidate. GMMU then determines the $64KB$ basic block to which the current eviction candidate belongs and then schedules the whole basic block for eviction and eventual write-back. Note that there can be pages in the basic block which were not accessed and just brought in by the prefetcher. All the 16 pages in the $64KB$ are written back as a single unit irrespective of the pages within are clean or dirty. This is because transferring memory in larger chunks improves PCI-e bandwidth and reduces latency instead of writing back multiple $4KB$ pages.

5.2 Tree-based Neighborhood (TBN_e) Pre-eviction

Our proposed tree-based neighborhood hardware eviction strategy is inspired by the TBN_p. It leverages the full-binary tree structures created and maintained for hardware prefetching at the time of managed allocation. Thus, it accounts for no additional implementation overhead. As discussed in Section 3.3, all nodes in these full-binary trees correspond to $64KB$ basic blocks and the root node of each tree corresponds to a maximum contiguous virtual space of $2MB$ large page or a size equivalent to $2^i * 64KB$. Like SL_e, an eviction candidate is chosen from the LRU list. Then a $64KB$ basic block, to which this eviction candidate belongs, is identified for pre-eviction. After selection of every pre-eviction candidate, GMMU traverses the whole tree updating the valid page size of all its parent nodes including root node by subtracting the size of the evicted basic block. At any point if the total valid size of any node is strictly less than 50% of the maximum valid size of that node, further pre-eviction decision is made by the GMMU which is in turn pushed down to the children till the leaf level. This process continues recursively till no more basic blocks can be identified for pre-eviction or no node higher than leaf level including root node has valid size less than 50% of the maximum capacity at the corresponding tree level. The eviction granularity in this scheme varies between $64KB$ to $1MB$ and thus it adapts between the two extremities of $4KB$ and $2MB$ eviction granularity for LRU.

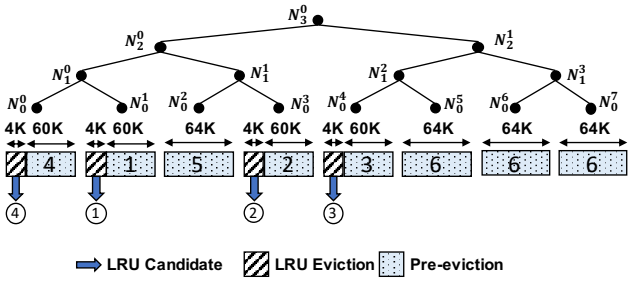


Figure 8: Demonstration of TBN_e on 512 KB memory chunk.

Figure 8 demonstrates the TBN_e on a $512KB$ memory allocation for simplicity. Initially, let us assume that all pages in this $512KB$ allocation are valid in the page table. Let us further assume that the first three entries in the LRU list correspond to the basic blocks N_0^1, N_0^3, and N_0^4. Upon over-subscription, when page replacement routine kicks in, these three basic blocks are identified for eviction one after another. After evicting the first three basic blocks, the valid size for each of the nodes N_1^0, N_1^1, and N_1^2 is updated to $64KB$

by GMMU. Further the valid sizes for nodes N_2^0, N_2^1, and N_3^0 are updated as $128KB$, $192KB$, and $320KB$ respectively. Let us now assume that the current least recently used page corresponds to the basic block N_0^0. After the fourth pre-eviction, the valid sizes of N_1^0, and N_2^0 are updated as $0KB$ and $64KB$ respectively. As the current valid size for N_2^0 is $64KB$ and is less than 50% of its maximum capacity, further pre-eviction decision is made for N_1^1 and is pushed to its children. This ultimately chooses N_0^2 as a pre-eviction candidate. At this point, GMMU traverses the tree and updates the valid sizes of all nodes in the tree. It then discovers the valid size of N_3^0 to be $192KB$ which is less than 50% of its maximum capacity. This pushes pre-eviction decision to N_2^1 and in turn to its children. This process identifies basic blocks N_0^5, N_0^6, and N_0^7 as pre-eviction candidates. As these blocks are contiguous GMMU groups them together into a single transfer.

5.3 Specific Design Choices

Both SL_e and TBN_e first select an eviction candidate from the LRU list and then identify the corresponding $64KB$ basic block for eviction. These basic blocks, up for eviction, can have some pages with dirty and/or access flags set in the page table along with some pages for which these flags are not set and only valid bits are set in the page table. We make a distinct design choice for how LRU page list is to be maintained in case of these pre-eviction policies. We place all the pages in the LRU when the valid flags of the corresponding page table entries are set in the GPU page table. This means LRU list contains all pages with valid flag set in the GPU page table in contrast to the traditional LRU list which only maintains pages with the access flags set in the page table. Further, a page is pushed to the back of the LRU list upon any read or write access in the course of execution. Upon evicting a basic block, all pages including the eviction candidate are removed from the LRU list. Hence, this design choice ensures all pages local to the eviction candidate are evicted irrespective of whether they are accessed or not. This is how SL_e and TBN_e deal with the unused prefetched pages migrated by the SL_p and TBN_p and free up contiguous virtual address space. We sort the pages first at large page level based on the access timestamp of the $2MB$ chunk they belong to. Then, within the $2MB$ large page, $64KB$ basic blocks are sorted based on their respective access timestamps. This hierarchical sorting ensures a global order at $2MB$ large page level and a local order of $64KB$ basic blocks at leaf-level of $2MB$ tree.

A known issue with LRU is that the performance degrades for a repetitive linear access pattern. For example, if there are N pages in the LRU page list, a CUDA kernel executing a loop over an array of $N+1$ pages will face a far-fault on each and every access. There have been a lot of research efforts invested in the past in modifying LRU to work with repetitive sequential access pattern as iterating over large arrays are common. One of such proposal is to switch to Most Recently Used (MRU) page replacement policy upon detecting such memory access pattern. However, detecting or predicting memory access pattern in runtime is itself a challenging problem and incurs large implementation and performance overheads. In this paper, we follow a simple solution to address this problem by reserving certain pages from the top of LRU page list such that they are not chosen as eviction candidates. Thus, reserving the top percentage of LRU page list reduces thrashing since the top percentage of pages in LRU list, which are chosen for immediate eviction, are also accessed first in the next iteration.

6 EVALUATION METHODOLOGY

In this section, we describe how we provide functional and timing simulation support for UVM and also the benchmarks to characterize and evaluate the design choices in UVM.

6.1 Simulation Framework

We extended GPGPU-Sim 3.x [4] to incorporate the control flow to resolve far-faults described in the Section 2.2. This enables modelling of on-demand paging and data migration. As mentioned in the Section 2.2, there are two components: far-fault handling latency and page migration to be considered while modelling turn around time to resolve a far-fault. GPU Technology Conference 2017 [23] mentions that page fault handling latency to be $30\mu s$. However, upon experimenting on real hardware with `GeForceGTX 1080 ti`, we found it to be $45\mu s$ on average. We mainly focus on the modelling of far-fault handling latency and PCI-e transfer latency in our simulator. Based on the Table 1, we deduce a function to express PCI-e bandwidth as a function of transfer size. In our simulator, we calculate PCI-e transfer latency based on this expression. We also consider an additional `100` core cycles for page table walk. The simulator makes simplified assumptions to model TLB and page table. Our TLB is roughly modeled after [22] proposed by Pichai et al. However, our TLB look up is performed in a single core cycle based on the assumption of fully-associative TLB. We use a multi-threaded model for page table walk as described in [3]. Along with on-demand paging, we incorporated the hardware prefetchers and eviction policies described in Sections 3 and 5 respectively in our simulator.

Table 2: Configuration parameters for GPGPU Simulator supporting Unified Virtual Memory

Simulator	GPGPU-Sim Unified Virtual Memory Smart
GPU Architecture	NVIDIA Pascal architecture
GPU Cores	28 SM, 128 cores each @ 1481 MHz
Page Size	4KB
Page Fault Handling Latency	**45** μs
Page Table Walk Latency	**100** core cycle
CPU-GPU Interconnect	PCIe 3.0 16x, 8 GTPS per channel per direction

We have also added functional simulation support for CUDA 8.0 UVM APIs- `cudaMallocManaged`, `cudaMemPrefetchAsync`, and `cudaDeviceSynchronize` to the simulator. This enables our simulator to run benchmarks written using the UVM APIs in addition to the timing modelling for CPU-GPU UVM. In addition to the functional and timing modelling of UVM, we have added an array of statistical counters to profile different aspects of UVM and analyze multiple design decisions involved. Our simulator models NVIDIA Pascal [19] like GPU architecture with the support for CUDA 8.0 APIs. Table 2 shows the additional configuration parameters that primarily enables timing simulation of UVM APIs on such architecture.

6.2 Application Suite

To evaluate different design decisions crucial to the success of UVM, we chose seven benchmarks from Rodinia [6], and PolyBench [10] benchmark suites. We have modified the benchmarks by replacing `cudaMalloc` API calls by calls to `cudaMallocManaged` and by removing all instances of `cudaMemcpy`. These chosen benchmarks exhibit diverse behavior: (i) intensive computation with iterative kernel launches, (ii) migrating pages once over the interconnect but repeatedly access them per iteration, (iii) access pages once but transfer multiple distinct pages over PCI-e, (iv) random page access pattern, (v) sequential, and dense page accesses over a small set of pages, (vi) sparse memory accesses over a large set of pages, (vii) streaming access pattern and etc. Due to impractically long simulation time for larger memory footprint, we have limited the working set size of these benchmarks ranging from $4MB$ to $38.5MB$ with an average memory footprint of $15.5MB$. We have published these modified benchmarks along with the simulator [8].

7 EVALUATION AND DISCUSSION

7.1 Pre-eviction Policies in Isolation

We have shown that the TBN_p has the best performance when device memory can accommodate the whole working set. So, for experiments in this section, we only consider TBN_p before over-subscription. Under over-subscription, the simulator disables hardware prefetcher and only migrates $4KB$ pages on-demand. This is because we want to investigate the sole impact of different eviction policies on the kernel execution time. Also for this experiment, we only consider working sets for the benchmarks being 110% of the device memory size. Figure 9 shows the result of our experiment.

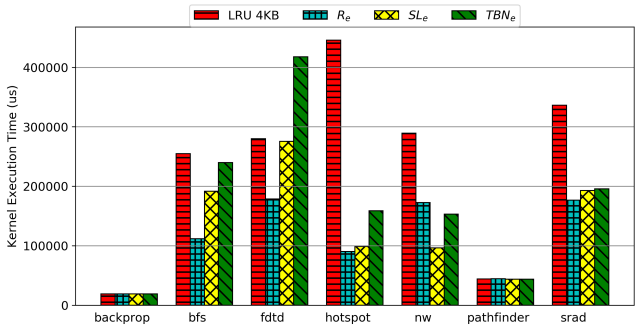


Figure 9: Comparing the effect of different eviction policies on kernel execution time. TBN_p is active before reaching device memory capacity. Upon over-subscription, hardware prefetcher is disabled and $4KB$ pages are migrated on-demand. Working set is 110% of the device memory size.

We see the following major behaviors exhibited by the benchmarks. `backprop` and `pathfinder` show no sensitivity to the choice of eviction policy. This is because both of these benchmarks exhibit streaming memory access pattern. Both of them scan a large vector in parts sequentially and do not reuse data across different iterations. However, for all other benchmarks, random eviction policy provides the best performance contrary to the popular belief that

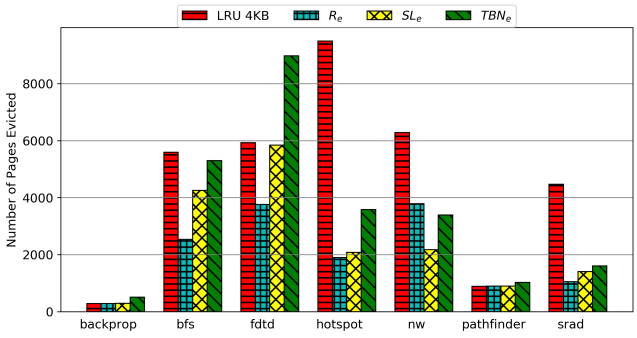


Figure 10: Comparing total number of pages evicted for different eviction schemes.

LRU and random page replacement policies have no performance difference [26]. Randomly picking a $4KB$ eviction candidate from the entire virtual address space reduces the chance of thrashing. In contrast, following LRU list increases thrashing for iterative kernels with data reuse. In Figure 10, we have plotted the total number of $4KB$ pages evicted by the different schemes. We can see that the kernel performance is highly correlated to the total number of pages being evicted by the corresponding page replacement policy as expected.

7.2 Combinations of Pre-eviction Policy and Hardware Prefetcher

To this end, we take UVM to the logical next step by pairing eviction policies and hardware prefetchers under over-subscription. The simulator enables the TBN_p before over-subscription. Also for the experiments in this section, we only consider working sets for the benchmarks being 110% of the device memory size. We chose 4 different combinations of eviction policies and page migration schemes such that they do not violate and rather respect each other's semantics.

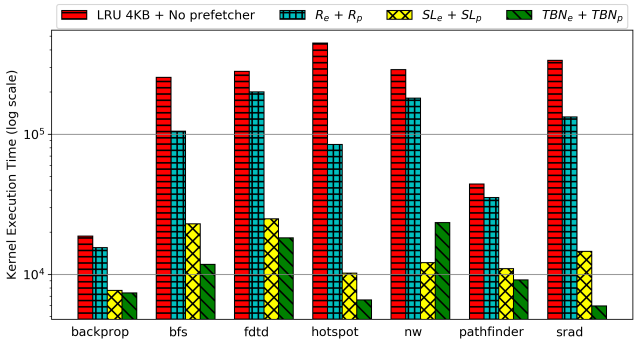
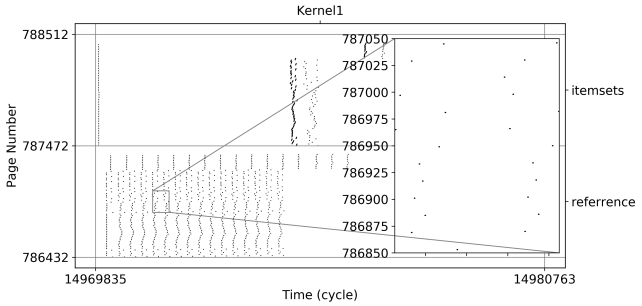
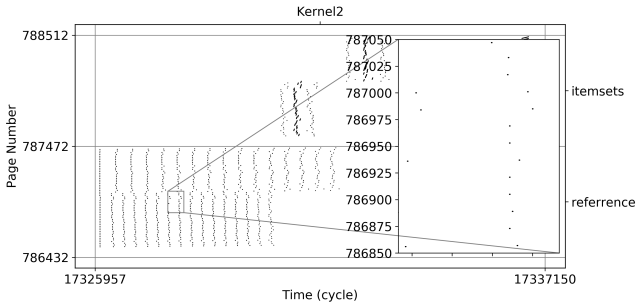


Figure 11: Comparing the effect of different combinations of eviction policies and hardware prefetcher after oversubscription on kernel execution time. TBN_p is active before reaching device memory capacity. Working set is 110% of the device memory size.

In Figure 11, we run the benchmarks to compare their kernel execution time for 4 settings under over-subscription: (i) LRU 4*KB* eviction and no hardware prefetching, (ii) R_e policy and R_p, (iii) SL_e and SL_p, and (iv) TBN_e and TBN_p. We see that the third and fourth combinations drastically outperform the first two. In particular, the combination of TBN_e and TBN_p provides an average 93% performance improvement compared to the combination of LRU 4*KB* eviction policy and 4*KB* on-demand page migration. This can be attributed to the improved PCI-e read and write bandwidth achieved by these combinations as they evict and prefetch memory in larger granularity other than 4*KB* which is the case for the first two combinations. Further, pre-eviction reduces the page access time by not waiting for pages to be written back and allowing prefetchers to prefetch pages reduces the number of page faults.



(a) Iteration 60



(b) Iteration 70

Figure 12: Page access pattern of nw benchmark without eviction.

One exception is nw. The combination of SL_e and SL_p yields better performance compared to the combination of TBN_e and TBN_p. To gain more insight in this behavior, we further analyze the memory access pattern of nw. In our example, nw runs for 127 iterations. Figure 12 shows the pages being accessed in iterations 60 and 70 (chosen randomly) respectively. The horizontal axis corresponds to the core cycle and the vertical axis shows the virtual page number. We can see that for nw, in every cycle, a set of pages, which are spaced far apart in the virtual address space, are accessed repeatedly over time. As the memory access is sparse yet localized and repeated over time, smaller granularity of eviction yields better performance than larger granularity. This is because evicting pages in larger chunk by TBN_e causes more thrashing than evicting 64*KB* basic blocks by SL_e.

7.3 Memory Over-subscription Sensitivity

In this experiment, we vary the percentage of memory oversubscription to study the scalability of combination of the proposed pre-veiction policy and hardware prefetcher. We use the combination of TBN_e and TBN_p after over-subscription for this experimental setup as this combination outperforms other combinations in general as seen in the previous section. Figure 13 shows that backprop, and pathfinder shows no sensitivity to memory over-subscription percentage as they exhibit streaming memory pattern. Other than nw, all other benchmarks scales up linearly. The order of magnitude performance degradation with higher percentage of memory oversubscription for nw can be attributed to its localized sparse memory access and large thrashing caused by the same.

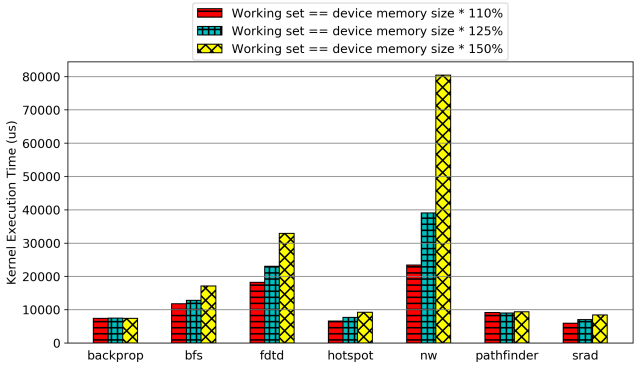


Figure 13: Sensitivity of combinations of TBN_e and TBN_p to the percentage of memory over-subscription by the working sets.

Note that, to simulate over-subscription, working sets of the benchmarks are not scaled, rather is controlled by a configuration parameter that specifies the device memory size in the simulation setup.

7.4 Reserving Percentage of LRU Page List from Eviction

To address the issue of page thrashing for benchmarks with data reuse over multiple iterations, we reserve a certain percentage of pages from the top of LRU page list from eviction as discussed in Section 5.3.

In Figure 14, we compare the kernel execution time of the benchmarks with the 10% and 20% reservation of LRU page list along with the combination of TBN_e and TBN_p against the same with no reservation. We see that streaming applications like backprop and pathfinder has no performance variation with LRU page reservation. The kernel performance improves with 10% reservation from the top of LRU list for all other benchmarks. However, with higher percentage of reservation, it hurts for certain benchmarks.

7.5 2MB Large Page Eviction

Based on the experimental results presented in the previous sections, we can conclude that pre-evicting pages in larger granularity

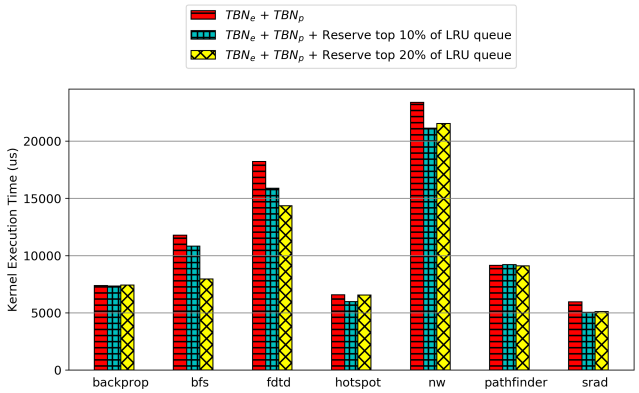


Figure 14: Effect of reserving a certain percentage of pages of LRU list from eviction on kernel runtime. Working set is 110% of the device memory size. TBN_p is active before reaching device memory capacity.

based on the spatio-temporal locality within 2*MB* large page enables further hardware prefetching under oversubscription and in turn provides better performance. We have also seen that 4*KB* LRU eviction renders hardware prefetching ineffective. So, a question can be asked then "Why not replacing pages in 2*MB* granularity?". Evicting 2*MB* large pages means invalidating the entire tree. This ultimately guarantees contiguous invalid pages required for the hardware prefetcher to work. Experiments on real hardware reveals that eviction granularity is indeed 2*MB* for NVIDIA GPUs. However, like aggressive prefetching, aggressive eviction is detrimental as it can cause serious page thrashing upon evicting highly referenced pages in case of repetitive kernel launch.

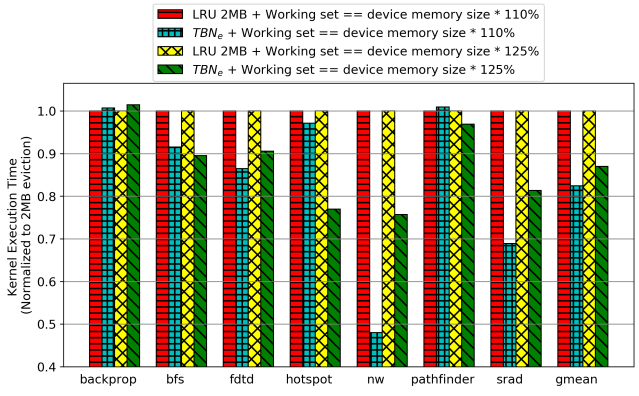


Figure 15: Comparing the performance of TBN_e against 2MB large page eviction.

In this experiment, we compare the TBN_e against the static 2*MB* LRU. Figure 15 shows that the TBN_e ensures an average 18.5% and up to 52% performance improvement compared to 2*MB* LRU under 110% memory over-subscription. By opportunistically determining a dynamic replacement granularity based on the current state of

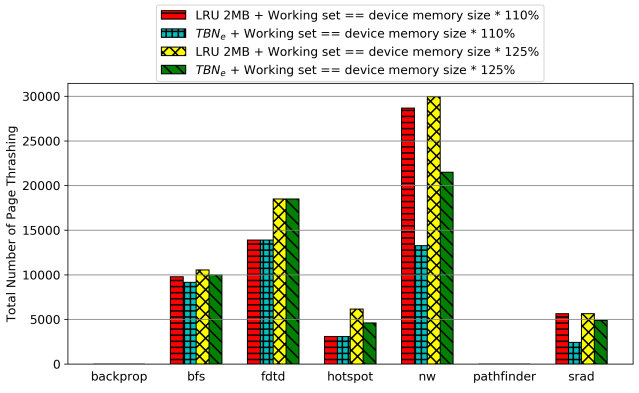


Figure 16: Comparing the effect of TBN_e and 2MB large page eviction on the total number of pages thrashed.

the 2*MB* full-tree, TBN_e navigates between the spectrum of 4*KB* and 2*MB* LRU eviction and overcome the limitations with both of these two extremes.

In the Figure 16, we show the average page thrashing caused by 2*MB* large-page eviction and TBN_e under 110% and 125% memory over-subscription. We can see that `backprop` and `pathfinder` shows no thrashing as they do not have any data reuse. For benchmarks like `bfs`, `hotspot`, `nw`, and `srad` the performance improvement by TBN_e compared to 2*MB* eviction can be attributed to the significant reduction in the number of page thrashing.

8 RELATED WORK

Unified Virtual Memory (UVM) support in modern discrete CPU-GPU systems [2, 19] has overcome many limitations present in the traditional "copy then execute" programming model [20, 21] by automating GPU memory management. Zheng et al [26] have first explored GPU's unified memory. Our simulation framework uses their replayable far-fault model to simulate on-demand paging. Researchers have investigated different techniques to reduce the overhead of address translation: TLB management [5, 7], and page table walk [9, 25]. However, far-fault handing latency and latency of data migration over PCI-e are the two major overheads incurred in UVM. Zheng et al [26] have proposed both user-directed prefetcher and user-agnostic hardware prefetchers to overlap data migration and kernel execution to hide these overheads. Latest NVIDIA GPUs [19] implement hardware prefetcher to ensure better performance.

Memory footprint of CUDA workloads has always been the limiting factor in GPU programming. Working sets are constrained by the size of GPU physical memory. Developers are forced to write smart programs to work around the memory over-subscription issue like in the VAST [15] runtime. VAST partitions data-parallel workloads based on available GPU physical memory. GPUswap [14] transparently relocates data from the GPU to system RAM under over-subscription. Agarwal et al [1] have proposed bandwidth-aware (BW-AWARE) page-placement policies for heterogeneous systems by programmer annotation after profiling data-structure accesses. All these techniques are plagued by huge performance

overhead. Li et al [16] has proposed a framework which requires hardware modification to characterize workloads before selecting a suitable strategy to address memory oversubscription. In this work, we propose new locality-aware pre-eviction policies. Our proposed pre-eviction policies are inspired by and thus compatible with the existing hardware prefetchers. Hence, they do not incur any additional implementation and performance overhead.

9 CONCLUSIONS

In this paper, we have reinforced the importance of hardware prefetchers in the success of Unified Memory when working set fits within GPU memory. On-demand paging enables computation over dataset larger than the physical memory capacity. In an over-subscribed memory situation, however, hardware prefetcher proves to be counterproductive. Our experiments show that naïve eviction policy can further contribute to the over-subscription issue. This necessitates the design of locality-aware eviction policies that are aware of the semantics of hardware prefetchers. To the best of our knowledge, this is the first work that introduces locality-aware pre-eviction policies that are compatible with hardware prefetcher. We analyzed memory access patterns of the workloads to gain more insight into the interplay of such pre-eviction policies and hardware prefetcher in UVM. Experimental results demonstrate that the proposed tree-based pre-eviction policy provides an average 93% and 18.5% performance speed-up compared to LRU based $4KB$ and $2MB$ page replacement strategies, respectively. The proposed scheme moves between two extremes of $4KB$ and $2MB$. By opportunistically determining a dynamic eviction size based on spatio-temporal locality within $2MB$ large page, it overcomes the limitations with page replacement strategies with fixed granularity. Moreover, as these pre-eviction schemes leverage the existing tree-based implementation of hardware prefetcher, they do not cost any additional implementation overhead. This makes this solution simple, pragmatic, and adaptable on real hardware irrespective of vendor-specific architectures.

ACKNOWLEDGMENTS

This material is based in part upon work supported by the National Science Foundation under Grant Number CCF-1725657. We thank the anonymous reviewers for their constructive feedback. We also thank Dr. Daniel Mosse for his feedback for improving the paper.

REFERENCES

[1] 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 Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 607–618. https://doi.org/10.1145/2694344.2694381

[2] AMD. 2017. Radeons Next-generation Vega Architecture. https://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf. (2017).

[3] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. 2017. Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 136–150.

[4] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 163–174.

[5] A. Bhattacharjee, D. Lustig, and M. Martonosi. 2011. Shared last-level TLBs for chip multiprocessors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 62–63. https://doi.org/10.1109/HPCA.2011.5749717

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 44–54.

[7] Jason Cong, Zhenman Fang, Yuchen Hao, and Glenn Reinmana. 2017. Supporting Address Translation for Accelerator-Centric Architectures. In *HPCA*.

[8] Debashis Ganguly. 2019. GPGPU-Sim UVM Smart. https://github.com/DebashisGanguly/gpgpu-sim_UVMSmart.git. (2019).

[9] Jayneel Gandhi, Arkaprava Basu, Mark D Hill, and Michael M Swift. 2014. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 178–189.

[10] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*. IEEE, 1–10.

[11] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. 2013. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 395–406.

[12] Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. 2013. Orchestrated scheduling and prefetching for GPGPUs. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 332–343.

[13] Teresa L Johnson, Matthew C Merten, and Wen-Mei W Hwu. 1997. Run-time spatial locality detection and optimization. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*. IEEE, 57–64.

[14] Jens Kehne, Jonathan Metter, and Frank Bellosa. 2015. GPUswap: Enabling Oversubscription of GPU Memory Through Transparent Swapping. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*. ACM, New York, NY, USA, 65–77.

[15] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. 2014. VAST: The illusion of a large memory space for GPUs. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*. IEEE, 443–454.

[16] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A Framework for Memory Oversubscription Management in Graphics Processing Units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 49–63. https://doi.org/10.1145/3297858.3304044

[17] Haikun Liu, Hai Jin, Xiaofei Liao, Wei Deng, Bingsheng He, and Cheng-zhong Xu. 2015. Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines. *IEEE Transactions on parallel and distributed systems* 26, 5 (2015), 1350–1363.

[18] NVIDIA. 2018. CUDA Runtime API - v10.0.130. https://docs.nvidia.com/cuda/cuda-runtime-api/. (2018). Accessed Sep 26, 2018.

[19] NVIDIA. 2018. NVIDIA Pascal Architecture. https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/. (2018). Accessed Sep 26, 2018.

[20] NVIDIA Corp. 2011. CUDA Toolkit 4.0. https://developer.nvidia.com/cuda-toolkit-40. (2011).

[21] NVIDIA Corp. 2014. NVIDIA GeForce GTX 750 Ti. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf. (2014).

[22] 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. *ACM SIGPLAN Notices* 49, 4 (2014), 743–758.

[23] Nikolay Sakharnykh. 2017. Unified Memory on Pascal and Volta. http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf. (2017). Accessed Sep 26, 2018.

[24] Nikolay Sakharnykh. 2018. Everything you need to know about Unified Memory. http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf. (2018). Accessed Sep 26, 2018.

[25] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H. Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. 2018. Scheduling Page Table Walks for Irregular GPU Applications *(ISCA '18)*.

[26] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 345–357.