GMT: GPU Orchestrated Memory Tiering for the Big Data Era

Chia-Hao Chang
cuc1057@psu.edu
The Pennsylvania State University
USA

Jihoon Han
jhan@psu.edu
The Pennsylvania State University
USA

Anand Sivasubramaniam
axs53@psu.edu
The Pennsylvania State University
USA

Vikram Sharma Mailthody
vmailthody@nvidia.com
NVIDIA Research
USA

Zaid Qureshi
zqureshi@nvidia.com
NVIDIA Research
USA

Wen-Mei Hwu
whwu@nvidia.com
NVIDIA Research
USA

Abstract

As the demand for processing larger datasets increases, GPUs need to reach deeper into their (memory) hierarchy to directly access capacities that only storage systems (SSDs) can hold. However, the state-of-the-art mechanisms to reach storage either employ software stacks running on the host CPUs as intermediaries (e.g. Dragon, HMM), which has been noted to perform poorly and not able to meet the throughput needs of GPU cores, or directly access SSDs through NVMe queues (BaM) which does not benefit from lower latencies that may be possible by having the host memory as an intermediate tier. This paper presents the design and implementation of GPU Memory Tiering (**GMT**) by implementing a GPU-orchestrated 3-tier hierarchy comprising GPU memory, host memory and SSDs, where the GPU orchestrates most of the transfers that are bandwidth/latency sensitive. Additionally, it is important to not blindly transfer pages from the GPU memory to host memory upon an eviction, and **GMT** employs a reuse-prediction based practical insertion policy to perform discretionary page placement/bypass. An implementation and evaluation on an actual platform demonstrates that **GMT** performs 50% better than the state-of-the-art 2-tier strategy (BaM) and over 350% better than the state-of-the-art 3-tier strategy that is orchestrated by host CPUs (HMM), over a number of GPU applications with diverse memory access characteristics.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04
https://doi.org/10.1145/3620666.3651353

ACM Reference Format:
Chia-Hao Chang, Jihoon Han, Anand Sivasubramaniam, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-Mei Hwu. 2024. GMT: GPU Orchestrated Memory Tiering for the Big Data Era. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620666.3651353

1 Introduction

With the incessant need to grow the data capacities accessible to a GPU, there has been considerable effort to extend a GPU's reach into the host memory, or even the storage device. Many of these efforts have only considered the former or the latter in isolation. Only recently has there been an attempt at a 3-tier hierarchy comprising the GPU memory, host memory and the SSD. These limited efforts [5, 31] have extended the 2-tiers (GPU + host memory) into the SSD regime, leveraging existing operating system support (page cache) on the host. As pointed out in another recent study [40], outsourcing data transfers needed by the GPU to the limited host cores is highly inefficient and cannot keep up with the GPU's throughput needs. However, there is no such GPU orchestrated 3-tier memory hierarchy today, which is the void that this paper intends to fill. At the same time, using conventional insertion/replacement techniques in the second tier, for pages evicted from the first tier, does not make the most effective use of its capacity. To address this, we propose a reuse-predicted strategy, discuss and implement the proposed design in a fully GPU-orchestrated manner on a real platform, and evaluate a wide-spectrum of applications with diverse characteristics. We demonstrate that our **GMT** system (Figure 1) is 50% (average) faster than the state-of-the-art GPU orchestrated 2-tier (GPU memory - SSD) hierarchy, and 357% (average) faster than the state-of-the-art host CPU orchestrated 3-tier hierarchy.

As application working sets grow, GPUs need to access much larger data capacities than what their memory can hold. Early GPU generations had a limited memory capacity on the GPU card, requiring software running on the host to identify and explicitly pre-load the required data on the GPU

memory, before starting the kernels/computations. This has a high overhead, and restricts how much computation can be offloaded at a time - requiring many more kernel offloads.

In the last few years, GPU vendors have provided a virtual memory - Unified Virtual Memory (UVM) in NVidia[1] and Shared Virtual Memory in AMD - that extends beyond the GPU memory, to enable accesses to portions of the host memory. To access such pages, a page faulting mechanism on the GPU is used to employ the drivers running on the host to explicitly move those pages to GPU memory (and subsequently remapped to avoid repeated faults). Still, these mechanisms only extend the reach of the GPU's address space to include the (GPU+)host memories. Further, software running on the host initiates these data transfers.

Recognizing the need for accessing datasets in the storage tier, there are prior works, such as GPUfs [45], Dragon [31] and ActivePointers [44], that enable GPU programming interfaces for such accesses. Of these, GPUfs attempts to implement a file system interface for these programs, while the others try to implement a virtual memory interface for specific structures/regions. Regardless of the interface itself, their underlying implementations still use software on the host CPU to perform these transfers between GPU memory and storage. Even the more general interfaces (e.g. Dragon [31], Heterogeneous Memory Management (HMM) [5]) simply extend the UVM capability into the storage/SSD spaces to extend the GPU reach. They integrate UVM with Linux's page cache but this mechanism still runs on the host core, i.e. CPU serves as an intermediary for GPU acccesses. There is also recent work [52] on application specific mechanisms, particularly for deep learning workloads, to allow spillovers of data beyond GPU memory. Unlike our motivation, such mechanisms are not general purpose to work across diverse workloads, especially those with irregular and/or dynamic memory access patterns.

While these advancements enable GPUs access terabytes of data on SSDs, well beyond the GB limits of GPU and CPU memories, these mechansims that rely on CPU orchestration do not scale when hundreds/thousands of GPU threads fault on their pages and request those simultaneously. This has been noted in a recent work, BaM [30, 39, 40], where the authors design and implement GPU-orchestrated transfers directly between SSDs and GPU memory. To circumvent the host, they allocate NVMe queues in GPU memory [32, 33], and map them via *nvidia_p2p_get_pages* and *nvidia_p2p_dma_map_pages*, making them visible to other devices on the PCIe bus. Through these memory mapped queues, GPU threads directly send NVMe I/O commands, which SSD controllers can act upon, without requiring the host (having a limited number of cores to become a bottleneck) as an intermediary.

BaM transfers only between the GPU memory and SSDs, ignoring the host memory that could potentially play an important role in lowering the latency (relative to SSDs) for some pages. This is the critical void that this paper intends to fill through **GMT** (GPU Memory Tiering) - a GPU orchestrated Tier-2 layer (host memory) in the GPU memory hierarchy, between the GPU memory Tier-1 and the SSD Tier-3. Towards this goal, this paper makes the following key contributions:

Contrib. #1: We propose leveraging the host memory as a Tier-2 layer in the hierarchy, between GPU memory and SSDs. Even though resident on the host side, placement/eviction/transfers in its management are directly orchestrated by the GPU (rather than any software on the host).

Contrib. #2: Even if this Tier can hold much more data than Tier-1, its capacity still needs to be carefully managed. Indiscriminately placing all pages evicted from Tier-1 into Tier-2, can end up evicting more useful pages from the latter. This is an additional dimension to the problem that most memory managers on the host side do not have to deal with. On the CPU side, the hierarchy is only 2-levels on most systems, and the operating system's choice of what to do when evicting a page from the top tier is rather clear.

Contrib. #3: We propose a placement algorithm, for victims from Tier-1, that tries to approximate Belady's OPT policy. The basic idea is to predict the remaining number of accesses (called Remaining Reuse Distance or RRD) to other pages before the candidate chosen as a victim from Tier-1 is accessed again. Based on this number, the page could either (i) continue in Tier-1, or (ii) placed in Tier-2, or (iii) simply thrown away if clean or (iv) written to SSD if dirty.

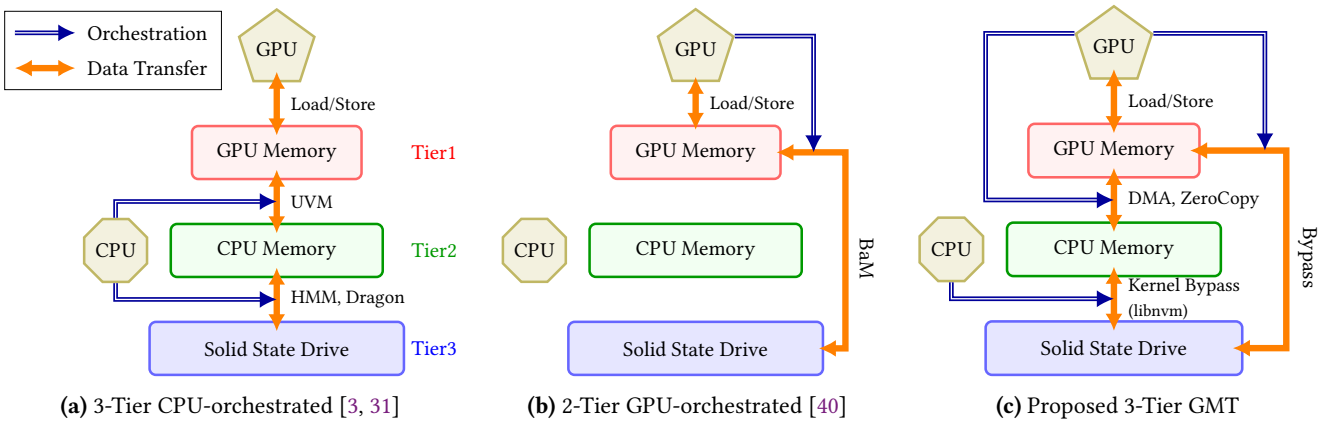
Contrib. #4: We propose a simple sampling based approach to predict RRDs, which can be easily done without overburdening the GPU, during early parts of an application's execution. Subsequently, it can use a simple 2-level history based prediction to figure out what to do upon a Tier-1 eviction. Further, we investigate how best to transfer data between the first 2 tiers for maximum effectiveness across different workloads.

Contrib. #5: We implement this `GMT-Reuse` mechanism, with 2 other simple Tier-2 based strawmen, on a NVidia A100 GPU plugged into a Xeon Gold 64-core CPU server running Linux 5.15.0 and NVidia 515.43.04 drivers, and evaluate our proposal with several GPU applications having diverse memory access characteristics.

Contrib. #6: We show that Tier-2 is extremely important in the hierarchy, with all three mechanisms leveraging this tier showing considerable speedups over BaM, which is the best performing tier-ing mechanism to date.

Contrib. #7: `GMT-Reuse` based Tier-2 incorporation provides 50% speedup over BaM on the average. While BaM [40] has

[1]Though this work is applicable across all GPU platforms, we use a NVidia GPU in our implementations and evaluations, and hence use NVidia terminologies in our discussions.



(a) 3-Tier CPU-orchestrated [3, 31] **(b)** 2-Tier GPU-orchestrated [40] **(c)** Proposed 3-Tier GMT

Figure 1. Memory Hierarchy for the GPU: Current Mechanisms and Proposed **GMT**

already been shown to outperform Dragon [31] a 3-tier hierarchy which is CPU-orchestrated, our results not only reiterate that, but show that the 3-tiered hierarchy managed by our GPU-orchestrated approach is 357% faster (than HMM [5]) across these applications. These benefits are significant across a range of Tier-2:Tier-1 capacities, and amplify as application working sets get larger.

2 GMT: GPU Managed 3-Tier Memory Hierarchy

Figure 1 captures (i) the 3 memory tiers available to a GPU, (ii) places the relevant prior work in perspective, and (iii) highlights our enhancements to build **GMT**.

There are fundamental differences in memory tiering for GPUs, compared to CPU's OS virtual memory mechanisms. First, unlike a 2-tier hierarchy dealt with in most systems, our context is a 3-tier setting. Adhering to inclusion and/or strict movement between successive tiers may not be the best option as we will show in our results. Second, data transfers are much more throughput sensitive - since the number of threads that may demand fault on a GPU is much higher than on the host CPU. This requires carefully orchestrated data transfer mechanisms - serially programming a single entity (e.g. DMA, or a few host cores), to perform this transfer as in a conventional system, does not meet the required throughput needs [40]. Finally, the SIMT nature of GPU operation can lead to considerable divergences/inefficiencies if we do not consider a coordinated effort at optimizing these, rather than deal with each thread's request individually.

In multi-tiering, strictly (i) moving data/pages up or down the tiers or (ii) adhering to the inclusion property, may not be the best option. Bypassing tiers can be beneficial based on application locality, as studied in other contexts (e.g. CPU caches [10, 18, 20, 22, 23, 25–27, 29, 46, 48, 50]), but not for these three GPU tiers. When a page is evicted from GPU memory, it may be better to directly evict it to the last tier (SSD) rather than the middle tier (CPU memory), if its next

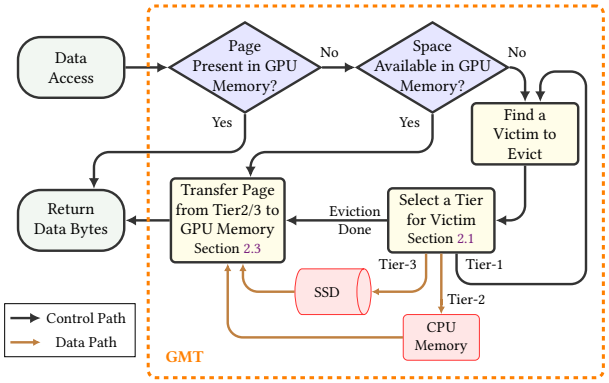


Figure 2. GPU thread Lifetime in **GMT**. Actions and required metadata, are maintained/performed by the GPU threads.

reuse would be later than a page that it would replace in the middle tier. Similarly, a page brought in from SSD, could skip the middle tier (and avoid evicting a more useful page), and directly be placed in GPU memory if it will be needed only for the very near future. Such choices can offer a richer set of options for page placement/replacement in order to more efficiently leverage the 3-tier memory hierarchy available to GPUs today. To our knowledge, this is the first work to do so, with this capability becoming more important as GPU datasets continue to grow in size, spilling beyond its own card memory and into the host memory and SSDs.

The best shown performance to date, for tiering GPU memory, is BaM [40] which only considers the SSD beyond GPU memory. Incorporating host memory as a Tier-2 structure into a mechanism like BaM requires answers to the following: - (i) when evicting a page from Tier-1, should we put it at all in Tier-2 (or in Tier-3 if dirty)? While reducing I/O, at the time of eviction and subsequent access, sub-optimal decision to place the page into Tier-2 may evict a more useful page in this Tier. Further, looking up Tier-2 to see whether a page is present, before going to storage introduces additional latency in the critical path. (ii) how do we efficiently transfer

pages between the tiers? Below, we cover options for the first question in Section 2.1 and discuss our implementation for the second question in Section 2.3. Before getting into these questions, below are some common parameters that we keep constant while designing **GMT**:

1. *Granularity:* Our unit for placement/movement is a 64KB page, which is the default size used by the UVM and prefetching mechanisms on Nvidia GPUs.
2. *When?* With several options for when to move pages - ahead of need (prefetching/pre-paging) or as a background activity - we mainly focus, as in BaM [40], on demand based movement, i.e. when pages are "demand-missed" on the GPU (movement into Tier-1, as in Figure. 2), which may result in the eviction of another page from Tier-1. Though not explicitly studied, placement options can also be considered in conjunction with prefetching of pages.
3. *What to evict from GPU memory?* We use the traditional clock-based replacement algorithm [37] (used in [40] as well), that offers an effective trade-off between approximating LRU and implementation efficiency.
4. *Bypassing in the up/down paths:* As explained earlier, bypassing tiers may allow more of the working set to be closer to the GPU cores depending on the application access patterns. For instance, a prior work [9] proposed bypassing the GPU memory in the context of UVM, if the page had only spatial locality which could be fulfilled by the GPU caches alone rather than additionally place such pages in GPU memory. BaM [40] automatically bypasses the host memory in both the up/down paths of flows in the hierarchy, since it was primarily built as a high throughput transfer mechanism between GPU memory and SSDs. Skipping the host tier in the "up"-path from SSD to GPUs may not pose a problem even if the decision was wrong. This is because such pages, upon eviction from Tier-1, can be selectively inserted into Tier-2 at a later point if needed (without necessarily coming in the critical path). Hence, in this work, similar to BaM, we bypass host memory in the "up"-path. However, the choice of whether to insert the evicted GPU memory page into CPU memory or into the SSD or possibly even thrown away if unmodified, can turn out to play an important role - it can evict a more critical page resident in host memory. This decision will be guided by the policies discussed next, implemented by GPU threads in **GMT**, compared to CPU orchestrated mechanisms in UVM [12], Dragon [31] and HMM [5].

2.1 Policies for Page Placement

We investigate three policies for dealing with a GPU memory page chosen by the clock algorithm as an eviction candidate.

2.1.1 `GMT-TierOrder`. This is the most natural option to consider, where each deeper level of the hierarchy holds the victim of the immediately preceding level. The clock replacement algorithm is used in both top tiers (GPU memory and CPU memory), with the victim of GPU memory going into CPU memory. This may lead to a subsequent eviction from CPU memory, where another clock algorithm decides the page to evict and write to SSD if dirty.

On the positive side, this scheme would keep working sets adhering to LRU/recency closer to the GPU - better than BaM which does not consider the CPU memory as a tier. However, its drawbacks are in (i) whether the recent past really reflects future behavior, (ii) cost of inserting (and any consequent eviction) into CPU memory as opposed to directly evicting (when clean) or writing into SSD (when dirty), and (iii) the additional cost of a replacement mechanism for host memory.

2.1.2 `GMT-Random`. Based on the negatives identified for the prior scheme, it is not even clear whether one should be very selective about where/whether to place each evicted page from GPU memory. To study this importance, we investigate this option where the choice is made randomly. Rather than evict only to SSD (when dirty as in BaM) or always to Host memory (as in TierOrder), a random choice is made to determine whether to place it in host memory or in SSD (latter only if dirty). This placement decision algorithm is again easy to implement. It can also hold much more data closer to the GPU (combination of GPU and host memory) than would be done by BaM. At the same time, it is unclear on how effective it would be in picking the right pages for closer placement over any approximate LRU (clock replacement) algorithm that `GMT-TierOrder` would pick.

2.1.3 `GMT-Reuse`. As is known from Belady's OPT algorithm [8], one should replace the page whose next reference (i.e. reuse) is furthest in the future, i.e. largest *reuse distance*. While LRU is meant to be a proxy for future reuse, this is not always the case and there have been alternate proposals to predict and implement reuse distance based policies, especially in smaller storage structures such as CPU caches [19, 24, 38, 43, 48, 49]. Below, we design a practical approach to estimating reuse distances at the point of Tier-1 page eviction for GPU memory tiering.

Overview: By default, we run the clock algorithm to find an eviction candidate from GPU memory. However, this candidate may have an earlier future reuse than another page resident in GPU memory. To address this, we estimate the distance, defined as the number of unique pages accessed, to the next access for this candidate, termed *Remaining Reuse Distance (RRD)* (see Figure 3), which is similar to the Estimated Time of Arrival concept in [43]. Rather than an exact value, we only need to predict the category that the RRD falls in, as per Eq. 1.

This categorization will determine whether the candidate should continue to remain in Tier-1 (short-reuse) or evicted to one of Host memory (medium-reuse) or SSD (long-reuse): If it falls in the "short-reuse" category, we will retain it in GPU memory and run another round of clock to find the next candidate and repeat. Once we find a candidate, we find

a free slot for it in host memory. In case, there is no free slot, we simply either discard (if clean) or put it in Tier-3 (if dirty). The rationale is that all pages in host memory, have been already predicted to have a reuse distance that is in the same equivalence class as the currently evicted one from Tier-1. Note that pages would move from Tier-2 to Tier-1, creating a free/vacant slot, upon a demand miss in Tier-1.

$$T(RRD) = \begin{cases} \text{short-reuse}, & \text{if } RRD < sizeof(Tier1) \\ \text{medium-reuse}, & \text{if } RRD \geq sizeof(Tier1) \\ & \quad \text{and } RRD < sizeof(Tier2) \\ \text{long-reuse}, & \text{if } RRD \geq sizeof(Tier2) \end{cases} \quad (1)$$

Implementation: While this may appear straightforward, the challenge is in predicting RRDs and classifying it into one of the 3 categories. This requires [16, 42, 47] not only future knowledge, but most practical implementations also require state maintenance for a page that needs to be updated on every access (even if it is not the page being accessed) [43]. There are attempts at such estimation for CPU caches, e.g. [43], where capacities are much smaller, states/counters to be maintained are small, and hardware counters can be exploited. These techniques cannot automatically scale for the large capacities in GPU and host memories, and the access parallelism that needs to be supported for GPUs, nor is there hardware available to readily perform some operations (e.g. virtual timestamping) on every access.

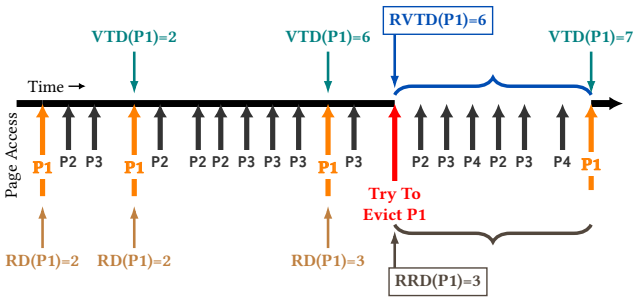


Figure 3. Reuse Distance (RD), Virtual Timestamp Distance (VTD), and Remaining RD (RRD), Remaining VTD (RVTD) at the time of eviction of page P1. Although the shown RRD and RVTD (in boxes) are exact, assuming future knowledge, in practice these will be predicted using prior history.

To this end, we employ a lightweight RRD estimation technique for these high throughput, large capacity systems. Inspired by [14], we use the Virtual Timestamp Distance (VTD, also known as non-unique reuse distance) as a proxy for reuse distances. *VTD of a page at any time is the number of (possibly non-unique) accesses since its last access.* We maintain a counter that is updated on each coalesced access (across threads of a warp). When a page is accessed, we timestamp that page with this counter's value. The VTD of the page is simply the difference from the counter's current value and its timestamped value. Note that the associated timestamp for a page is not needed/modified until the same page is accessed again, and can also benefit from available hardware counter implementations. See Figure 3 for an example. When a page (P1 in example) comes up for eviction, if we can predict its VTD upon its next reference (7 in this example), then we can calculate what we refer to as RVTD (Remaining VTD) - by subtracting the current timestamp (at eviction time) from the VTD of the page's next reference. There are 2 steps to RRD prediction of a candidate page detailed next. Eventually, knowing what tier (i.e. equivalence classes Eq. 1) to place a candidate page in is more important than exactly estimating RRDs/VTDs.

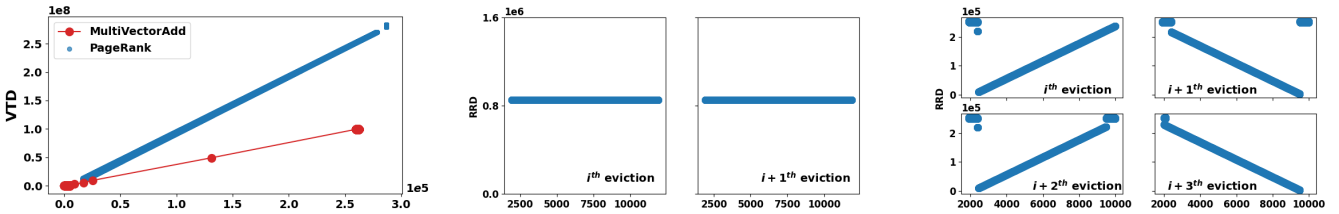
Step 1: Given current RVTD of a candidate, what is its estimated RRD? Taking 2 representative applications, MultiVectorAdd and PageRank, we instrument each access to obtain actual RDs and VTDs - this is an instrumented run that is used to merely demonstrate the rationale for our approach, even though it is infeasible to do this for all applications and all the time. The resulting correlation between the two is plotted for all pages in Figure 4a. Even though this is shown specifically for 2 applications, similar observations can be made across others. There are two important observations in this graph: (a) there is a good correlation (linear in fact), between VTD and reuse distances - implying we could use the former as a proxy for the latter in estimations; and (b) even though this linear relation is observed for accesses to all pages as an aggregate, the linearity also holds for randomly selected pages amongst this aggregate - implying that if we could infer the linear relationship for a small sample of the pages, that may suffice to extrapolate for other pages.

Predicting RRD from VTDs of a small sample: Based on the observations from Figure 4a, if we assume such a linear relationship between reuse distance and VTD, we can express this relationship as the following linear equation relating $VTDs$ and RD (and between $RVTD$ and RRD) as below with slope m and offset b

$$RD = m * VTD + b \quad (2)$$

$$RRD = m * RVTD + b \quad (3)$$

Such an equation would help us project the RRD for any given RVTD during the execution. In the above, the linear equation (regression) has been done by exhaustively running the entire instrumented application at a high cost. However, in reality, one does not wish to do so, given the high cost of doing this, defeating the entire purpose. Instead, we propose to (a) collect a small sample of (VTD, reuse distance) pairs early in the execution, and (b) use these to produce the linear equation with regression. While (a) is relatively simple, it is done directly on the GPUs. However, (b) can be more expensive and we offload the regression to the CPU, to avoid stalling the GPU threads progressing on normal application



(a) VTD (y-axis) vs. Reuse Distance (x-axis)

(b) MultiVectorAdd: RRD at Tier-1 Evictions (x-axis: Page Number, y-axis: RRD)

(c) PageRank: RRD at Tier-1 Evictions (x-axis: Page Number, y-axis: RRD)

Figure 4. RRD Characteristics of MultiVectorAdd and PageRank

execution. For this purpose, the GPU pushes collected VTD samples into a queue shared with the CPU, that is regularly consumed by a dedicated thread on the latter. This thread uses these samples and employs a tree-based method [13, 17] to calculate actual reuse distances from the VTDs. Once these "training" pairs of (VTD,reuse distance) are obtained, the CPU performs an Ordinary Least Squares (OLS) regression on those samples to get coefficients, slope m and offset b, of the linear relation. We could proceed with a default strategy (`GMT-Random` or `GMT-TierOrder`) until we collect enough samples (typically we collect hundreds of thousands) to get the regression equation. However, rather than wait until we get this final equation at the end of sampling, we pipeline the samples (every 10000 samples) to the CPU thread, which iteratively improves on the regression from the prior set of samples and provides back an updated m and b for the GPU. We find that this approach results in better placement for the early part of the execution.

Step 2: How do we predict current RVTD from prior history?

We still have not tackled the harder problem of predicting the (future) RRD of the current eviction candidate from Tier-1 upon its next access. To do this, we again draw inspiration from real behaviors in applications. Consider the actual (this is again drawn from fully instrumented runs with a post-mortem analysis) RRD distributions of pages, when they are evicted from Tier-1. In `MultiVectorAdd` (Figure 4b), we see that a page has the same RRD each time it is evicted from Tier-1, i.e. we can use the actual RRD (or RVTD because they are correlated) from the (i-1)-th eviction to predict the RRD for the i-th eviction. Consider another application, `PageRank` (Figure 4c), where though there is a correlation in RRDs from prior evictions, the pattern is not constant - it alternates. In the interest of space, we do not plot such figures for all the applications, but do note that historical trends of RRDs of a page from prior evictions can be useful to predict the current RRD when a page is a candidate for Tier-1 eviction. While we could build deep histories, a simple 2-level history suffices for making fairly accurate prediction. We keep track of the tiers that a page should have been placed in "correctly" upon its 2 prior evictions from GPU memory, and use this to implement a 3-state Markov chain, as shown in Figure 5.

Each state in this chain represents the "correct" tier that this page should have been placed in, upon its prior eviction from GPU memory. Note that this can be found out when a page is brought into GPU memory, since we know its exact RVTD/RRD from its prior eviction, and can use Eq. 1 to find the correct tier it should have been placed in. Once we know the correct tiers upon GPU memory eviction for a page, we can use this to update the transition weight between the 2nd last and immediately prior eviction states. This update is done whenever the page is brought into GPU memory. When the page next comes up for eviction, we can simply look at its last "correct" tier (state), compare the 3 transition weights coming out of this state, and use that to decide which tier we should next place this page in. Maintaining this state takes negligible space for each page.

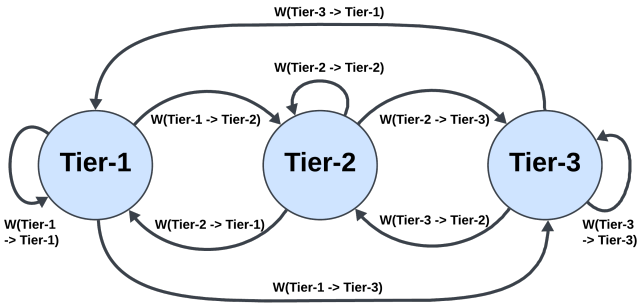


Figure 5. Proposed 3-state Markov Chain Predictor. (W(Tier-X -> Tier-Y): transition weight from Tier-X to Tier-Y)

2.2 Placement and Eviction in Tier-2

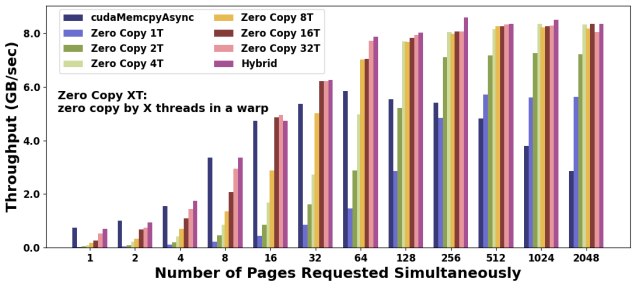
When placing an evicted page from Tier-1 into Tier-2 in these policies, we need to decide where to place it (and consequently what to evict from Tier-2). As long as there is an empty slot, we can simply place it there without disrupting other pages. Even after Tier-2 fills up, an empty slot can show up without an explicit eviction if the GPU references this page, moving it to Tier-1 (we do not duplicate pages across Tiers 1 and 2). If there is no such empty slot, then we evict a page using a simple FIFO mechanism in Tier-2.

In GMT-Reuse, there could be an interesting scenario based on application characteristics - if an overwhelming number of recent Tier-1 evictions are predicted to be placed in Tier-3, the host memory could become largely under-utilized. One may as well use this, relatively (compared to SSD) lower latency tier for some pages, instead of placing them all in Tier-3. For instance, consider a nested loop accessing a large array where successive accesses to an element are widely apart with RRD falling in the Tier-3 range - in this case, Tier-2 would be largely unfilled and it would make sense to place some of the (predicted Tier-3) pages in Tier-2. We implement a simple heuristic for such scenarios - if greater than 80% of the last evictions from Tier-1 have an RRD that would place the pages in Tier-3, we still place the current eviction into Tier-2 even if the prediction asks us to place it in Tier-3.

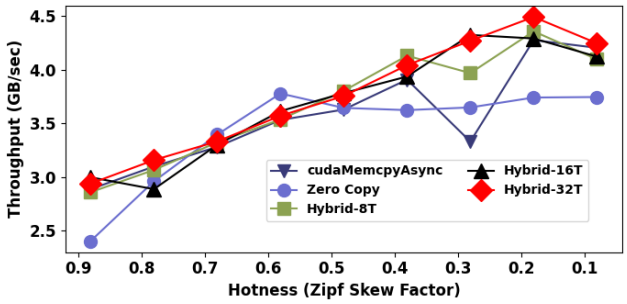
2.3 Mechanisms for Page Transfer

There are 3 pairs of entities/tiers between which pages need to be transferred in **GMT**:
(CPU memory - SSD): This is not unique to **GMT**, and is not in the critical path of GPU accesses. We use conventional user-space I/O (using libnvm) for these transfers (see Figure 1).
(GPU memory - SSD): We use the recent open-source GPU-direct mechanism, BaM [40], to perform these transfers.
(GPU memory - CPU Memory): There are two possible methods to transfer between these tiers: (i) via cudaMemcpyAsync, where the DMA is employed to move data between them, initiated by a single GPU thread; and (ii) several GPU threads (typically in a warp) directly employ load-store instructions on pinned CPU memory (zero-copy) to move the data as in [35]. There are trade-offs between these methods, as noted in [35], and their performance is shown in Figure 6a. Directly employing numerous (in fact, the entire warp of 32) GPU threads can sustain much higher throughput for a large number of *non-contiguous* transfers, compared to the DMA mechanism which becomes a serialization bottleneck in cudaMemcpyAsync. At the other end, for smaller number of non-contiguous transfers, the benefits of transfer efficiency with zero-copy is not sufficient to amortize the overheads of pinning these pages (to avoid replacement) before the zero-copy is performed.

These results suggest a Hybrid option which uses cudaMemcpyAsync for a smaller number of pages, and zero-copy otherwise. However, even when there are numerous pages to transfer, zero-copy may not be efficient if we cannot employ a lot of threads for the transfer. We illustrate this by implementing Hybrid-XT, which uses zero-copy only when (a) the number of pages to be transferred exceeds 8 (cross-over point in Figure 6a), and (b) we can employ at least "X" threads in a warp for these transfers. We run a microbenchmark where all GPU threads repeatedly generate page addresses drawn from a zipf distribution [36]. The skewness of the distribution is varied from 0 to 1 - controlling how many unique pages are



(a) Transfer Efficiency for Non-Contiguous Pages



(b) Delivered Bandwidth for Zipf Accesses to Pages

Figure 6. Comparing Transfer Schemes between Tiers 1 & 2

requested (higher skew implies fewer distinct pages) - to capture diverse application locality characteristics. Results are shown in Figure 6b for these Hybrid-XT schemes (X=8,16 and 32), along with always resorting to the zero-copy or cudaMemcpyAsync mechanisms. Skewness closer to 1.0 (left side of x-axis) will involve fewer transfers compared to the right side, similar to the x-axis of Figure 6a. Across this wide range of skewness, we see that Hybrid-32T - *using zero-copy only when number of page transfers is at least 8, and the entire warp of 32 threads can be employed for the transfer* - does (or is close to) the best. Hence, we use Hybrid-32T for CPU-GPU memory transfers in **GMT**.

3 Evaluation

3.1 System and Baselines

We have implemented **GMT** with all 3 policies on a server system shown in Table 1. As a baseline, we use the previously proposed BaM substrate that has been shown to provide maximal (saturating the underlying SSD/PCIe peak bankwidths) throughput between GPU memory and tertiary storage. Note that BaM uses only 2 tiers - the GPU memory and SSDs - not leveraging the host memory as an intermediate tier. These evaluations will demonstrate the importance of leveraging this intermediate tier, along with the need for a reuse-aware mechanism for page placement as in GMT-Reuse, rather than simply move pages across tiers.

In later experiments, we also compare **GMT** with a 3-tier placement mechanism - Nvidia's recent implementation of

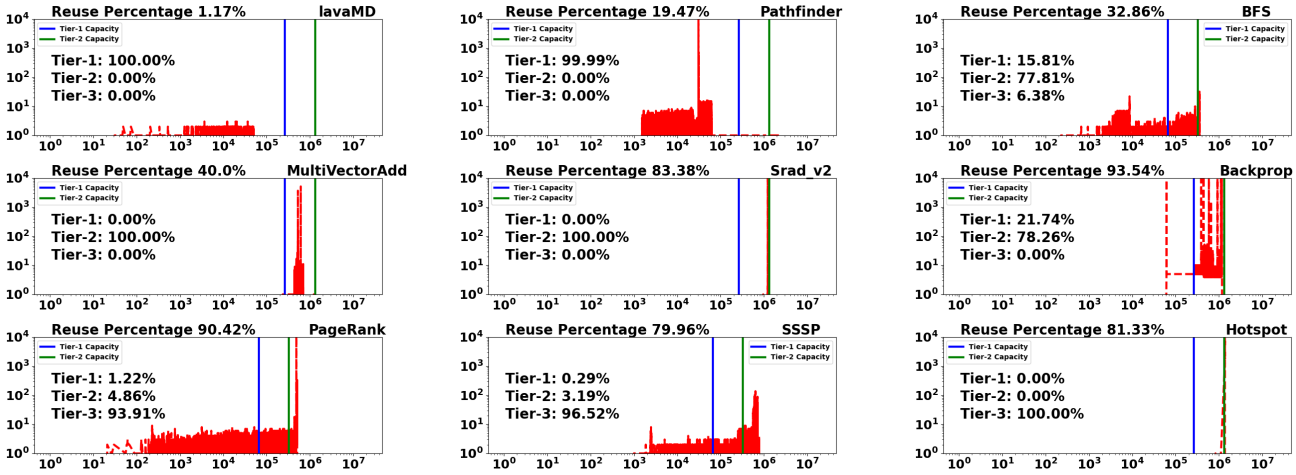


Figure 7. Remaining Reuse Distance Distribution (y-axis: frequency, x-axis: RRD)

Configuration	Specification
System	TYAN B7119F83V8E4HR-2T-N
CPU	Intel Xeon(R) Gold 6226 64-CPU
GPU	Nvidia A100-40GB PCIe
DRAM	256 GB DDR4
SSD	Single Gen3 x4 Samsung 970 EVO Plus
Interconnect	PCIe Gen3 x16
Kernel/Nvidia driver	v5.15.0 / v515.43.04

Table 1. System Specification

Linux HMM [5] which extends UVM to the tertiary SSD storage via the host-based Linux paging system/cache. Unlike our proposals, or BaM, this mechanism requires the CPU as an intermediary to manage this hierarchy.

Without loss of generality, we cap the GPU memory (Tier-1) capacity to 16 GB - a power of 2 is more flexible to tweaking the over-subscription factor that exercises the 3-level hierarchy [2]. By default, we set Tier-2 to be 4X larger than Tier-1, similar to other works on tiered memory [28, 34, 41], and the over-subscription factor to 2. We conduct subsequent sensitivity experiments to vary these parameters.

3.2 Workloads and their Characteristics

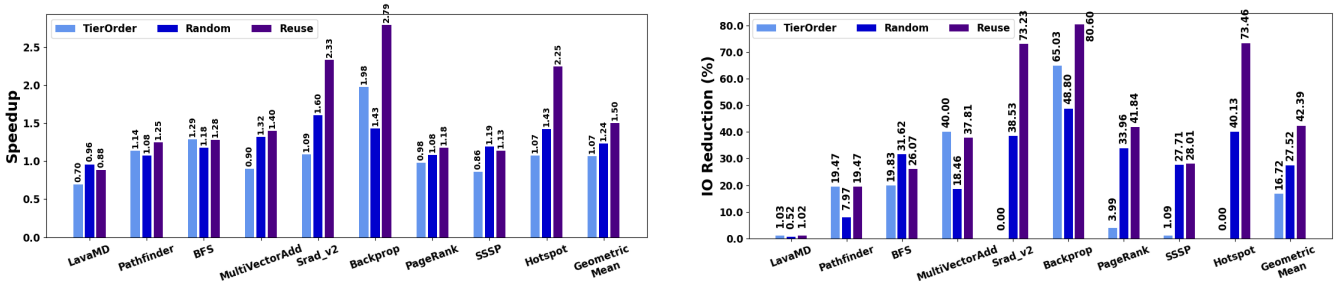
We use nine applications (Table 2) that have also been used in related studies [31, 40] for evaluations, which are modified as in [40] to leverage the tiers. These include applications with regular (statically defined) and irregular (dynamic/pointer/-graph [7]) access patterns.

Two factors impact an application's performance in the memory hierarchy: (i) its reuse of data - if there is little/no reuse, there is little need for a multi-level hierarchy; and (ii) the reuse distances themselves. Even if there is considerable reuse, if the distances are (a) very small (to fit in GPU memory itself), the hierarchy would not help much; or (b) very large (exceeding the GPU+Host memory capacities), the data is more likely to be in the SSD making it more I/O-bound

[2]We define the Over-subscription factor as the ratio of the application's working set size to the sum of capacities of GPU (Tier-1) and CPU (Tier-2) memory.

Application	Description	Reuse % of a Page	Total I/O (GB)
LavaMD	Particle simulation, neighbor accesses (Rodinia)	1.17%	168
Pathfinder	Dynamic programming, row-by-row iter. (Rodinia)	19.47%	202
BFS	Graph algorithm, data-dependent vertex/edge accesses (BaM) (source graph: *GAP-Kron*)	32.86%	87
Multi-VectorAdd	Linear algebra. Output vector repeatedly accessed (BaM [40])	40.0%	267
Srad	Image processing, 4 grid neighbor accesses (Rodinia)	83.38%	270
Backprop	Machine learning algorithm, layer-by-layer forward pass and backward propagation (Rodinia [11])	93.54%	6823
PageRank	Graph algorithm, data-dependent vertex/edge accesses (BaM) (source graph: *GAP-Kron* [15])	90.42%	349
SSSP	Graph algorithm, data-dependent vertex/edge accesses (BaM) (source graph: *GAP-Kron*)	79.96%	239
Hotspot	Thermal simulation, iterations on a grid (Rodinia)	81.33%	1492

Table 2. Applications (and their sources)



(a) Speedup compared to BaM

(b) Reduction in I/O over BaM

Figure 8. Results for Tier-1=16GB, Tier-2=64GB and Oversubscription Factor = 2

(than leveraging the memory hierarchy). To understand the behavior of these applications, in Figure 7, we show both the percentage of pages that are actually reused in an application (above the graph of each application), as well as the distribution of RRDs that was defined in section 2.1.3. In these graphs, we also plot vertical lines demarcating the GPU memory capacity, and the GPU+CPU memory capacities, where these ranges capture what should ideally be placed/fulfilled from a particular tier. As can be seen, the selected applications span the spectrum - from `lavaMD` with reuse percentage as low as 1% to `Backprop` with reuse as high as 93.5%. Further, the working set of these applications range from those which are small enough to almost entirely fit in GPU memory (`BFS` and `lavaMD` to those which have very poor temporal locality (as in `Hotspot`), and the others fall in between.

3.3 Workload Impact on Importance of Tier-2

Figure 8a shows the speedup of **GMT** policies, over the prior BaM that only uses 2 Tiers. While all policies show speedup over BaM, `GMT-Reuse` shows a significantly higher average speedup (1.5) over the other two - 1.07 and 1.24 for `GMT-TierOrder` and `GMT-Random` respectively. These results point out the importance of both: (i) leveraging the intermediate (host memory) tier in the GPU memory hierarchy, as opposed to only moving pages between GPU and SSDs, thus reducing I/O (shown in Figure 8b); and (ii) being more discretionary about what pages to place in the 2nd tier (upon eviction from first) as in `GMT-Reuse` (whose prediction accuracy is shown in Figure 9), rather than always placing upon eviction (`GMT-TierOrder`) or randomly (`GMT-Random`).

We next explain the variance across the applications using their characteristics in Figure 7. We categorize them based on their page reuse behavior and RRD bias (i.e. tier where the bulk of RRDs lie when pages are evicted from Tier-1).
Low Reuse, Tier-1 Bias: Consider `lavaMD` where the reuse percentage is very low (1.17%). Even this low reuse is heavily concentrated in the Tier-1 GPU memory (see Figure 7) accesses, leading to very few accesses trickling down to even

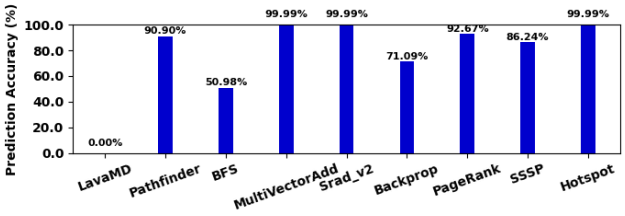
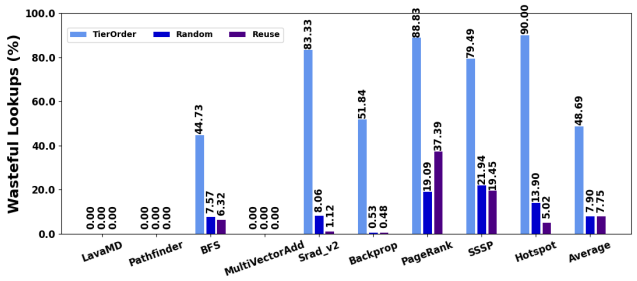


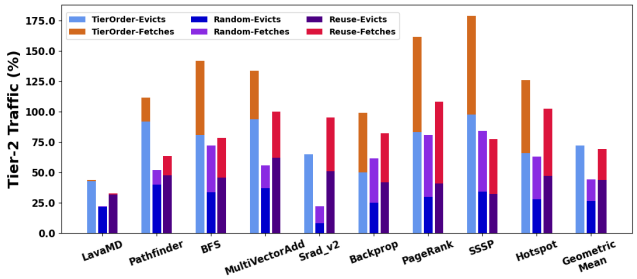
Figure 9. Prediction Accuracy of `GMT-Reuse` for Results in Figure 8

Tier-2. Consequently, the policies which are intended to use the host memory, are not able to produce much speedup over BaM. In fact, `GMT-Random` (in `lavaMD`) is even outperforming the proposed `GMT-Reuse`. This is because there is hardly any history built-up for making predictions (see Figure 9) in `GMT-Reuse`, which can end up making a poor placement choice. Still the performance is not very different, given the few accesses trickling down to deeper levels of the hierarchy.

Even though `Pathfinder` falls in this category with an overwhelming 99.99% of RRDs falling within Tier-1, it has a slightly higher reuse percentage (19.47%) compared to `lavaMD`. Consequently, its speedup with `GMT-Reuse`, as well as the other two policies, does turn out higher (25% over BaM) compared to `lavaMD` which suffers a slowdown of 12% (due to mispredictions without a long history).
Medium Reuse, Tier-2 Bias: Consider `BFS` and `MultiVectorAdd` where reuse percentages are relatively higher (32.86% and 40%), and a large number of these reuse RRDs fall in the Tier-2 category. This not only favors `GMT-Reuse` which provides 28% and 40% speedups in these two applications, but also for `GMT-Random`, which shows good performance benefits over BaM (which does not take advantage of Tier-2). `GMT-TierOrder` also does quite well for `BFS`, but in the case of `MultiVectorAdd`, with larger reuse distances than **BFS**, newly inserted pages into Tier-2 evict pages that will be least-furthest in the future to be accessed, which we know from Belady's algorithm need to be given priority. This is the



(a) Tier-2 Wasteful Lookups



(b) Evictions from Tier-1 Placed into Tier-2 and Fetches from Tier-2 placed in Tier-1

Figure 10. Overheads due to Tier-2 (Tier-1=16GB, Tier-2=64GB). `GMT-Reuse` reduces wasteful Tier-2 lookups, without significantly adding to the PCIe traffic due to Tier1 - Tier2 transfers, while maximizing Tier-2 utilization

usual problem of FIFO or LRU for cases where the working sets become exceedingly large.

High Reuse, Tier-2 Bias: `Srad` and `Backprop` have these desirable characteristics to benefit from the CPU-tier. We can see clear evidence of this in their speedups where `GMT-Reuse` gives speedups of 133% and 179% respectively. The considerable gap between `GMT-TierOrder`/`GMT-Random` vs. `GMT-Reuse` points to the importance of being selective about what to place in Tier-2. Further, we note that our RRD predictor is also fairly high in these applications, giving further credibility to our heuristic. These high speedups stem largely from the big reductions in SSD I/O (73% and 81%) due to high hit rates in host memory.

High Reuse, Tier-3 Bias: In `PageRank`, `SSSP` and `Hotspot`, reuse percentages (90%, 80% and 81% respectively) are high. However, the RRDs for these fall heavily in the Tier-3 category (94%, 97% and 100% respectively). Based on these characteristics, one would think that our Tier-2 leveraging policies will not be beneficial in these applications. However, when we examine their speedups for `GMT-Reuse` we see benefits of 18%, 13% and 125% in these 3 applications. While to some extent, the RRDs for Tier-2 placement are non-zero in the first 2 applications, it is nearly zero for `Hotspot` which would make it even more non-intuitive as to why its speedup is so high. With very large RRDs, as was explained in Section 2.2, if one were to simply place the page in the predicted tier, nearly all pages would go to Tier-3 and there will be a gross under-utilization of Tier-2. Our proposed enhancement which recognizes this behavior and forces at least some of these pages down to Tier-2 helps reduce the number of SSD accesses (e.g. 73% reduction for `Hotspot`). These results point to the importance of leveraging the host memory in the GPU memory hierarchy and `GMT-Reuse`'s ability to exploit it, even when reuse distances are very large.

3.4 Placement Accuracy Impact

There are trade-offs when introducing a Tier-2 host memory between the GPU memory and SSD, with the benefits/costs depending on the accuracy of placing an evicted page from Tier-1 into Tier-2 or Tier-3.

Benefits: Retrieving a page from host memory is faster (around 50 μs) than retrieving it from the SSD (around 130 μs). Consequently, the reduction in I/O in Figure 8b with the **GMT** policies, contributes significantly to their speedups over a mechanism like BaM which does not leverage host memory.

New Costs: As in any multi-level hierarchy, introducing Tier-2 between the GPU memory and SSD can add overheads:

1. BaM can directly initiate I/O (SSD) upon GPU page misses. However, all our 3-tier strategies, involve an intermediate (by the software on GPU) look up of Tier-2, which if successful would be considered a "useful" lookup. If unsuccessful, this lookup adds to latencies (around 50 ns) in the critical path before going to the SSD, which would be less efficient than a BaM-like mechanism that avoids this tier. It could also lead to extra work on the GPU cores, which may have otherwise performed useful application work. Figure 10a plots such wasteful lookups for pages in Tier-2 as a percentage of page misses in Tier-1. As shown, `GMT-Reuse` has the fewest unnecessary lookups because of its higher prediction accuracy. `GMT-TierOrder` does quite bad on this metric - if reuse distances are large, always pushing it down to the next tier will not be fruitful on the next lookup since it would have been evicted by then. `GMT-Random` is not really a bad choice from this perspective.
2. Tier-2 based policies may involve more PCIe bus transfers, even if they require fewer transfers to/from SSDs. This is because the same page may move between Tiers 1 and 2 multiple times, after each time it is brought in from the SSD. Figure 10b plots the number of such host-GPU memory transfers over the execution of an application, as a percentage of GPU-SSD transfers in BaM. In general, for each bar in this figure, we want the top part to roughly equal the bottom part, i.e. the number of placements we make into Tier-2 match the number of retrievals, to indicate that these placements are correct decisions and are being reused later. In the results, we first note that `GMT-TierOrder` is quite

poor, not just in the former outweighing the latter (i.e. poor placement decisions) in many cases but also introduces a lot more traffic on the PCIe bus. While GMT-Random does better than GMT-TierOrder in wasteful lookups and PCIe traffic, the number of evicts (which are hits in Tier-2) is much lower than in our GMT-Reuse approach. On the average, we see the top and bottom parts of the bars matching more closely for GMT-Reuse.

On the average, these costs amount to around 2.41% for an application. However, I/O reduction benefits far outweigh these costs, to give speedups for all proposed **GMT** polices. Of the 3 policies, GMT-Reuse further optimizes for the hits in Tier-2, reducing the I/Os incurred much more than the other two mechanisms (by 25.67% and 14.87% over GMT-TierOrder and GMT-Random), and the number of redundant lookups in Tier-2 (by 40.94% and 0.15% over GMT-TierOrder and GMT-Random). These results reiterate the importance of including the host memory in the GPU memory hierarchy, and the criticality of improving the hit rate to this tier.

3.5 Sensitivity Analysis

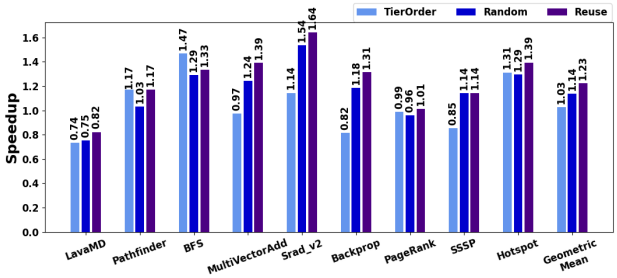


Figure 11. Speedup over BaM for Oversubscription Factor of 4 (Tier-1=16GB, Tier-2=64GB)

Over-subscription factor: To test sensitivity to larger datasets, we run experiments by doubling (to 4) the over-subscription factor. This was achieved by doubling the dataset size for non-graph applications, and reducing the Tier-1/Tier-2 capacity by half for graph applications. A higher over-subscription would lead to more I/O in the Tier-2 leveraging schemes, belittling their benefits over BaM. Figure 11 shows the resulting speedups over BaM. While absolute speedups do decrease as expected (1.23 for GMT-Reuse, 1.03 for GMT-TierOrder and 1.14 for GMT-Random), GMT-Reuse's speedup over BaM is still considerable, reiterating the need for utilizing the Tier-2 capacity wisely than is done in the other 2 policies.

Tier-2:Tier-1 capacities: The above experiments use 16 GB of GPU memory and 32 GB of host memory, i.e. 2:1 ratio. In the future, while both capacities are expected to grow, the former may grow faster than the latter - since it is also used by the host, and it conforms to more commodity/cheaper DDR roadmaps. Hence, we have varied this ratio and studied the

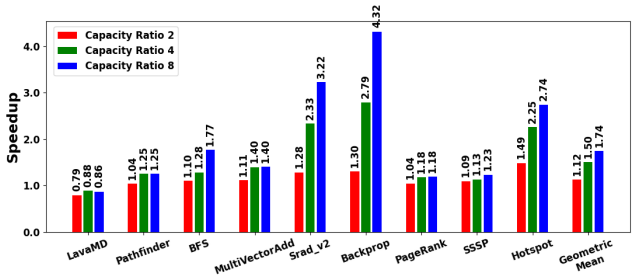


Figure 12. Speedup of GMT-Reuse over BaM with Different Capacities of (Tier-1:Tier-2). Ratios = 2 (16GB, 32GB); 4 (16GB, 64GB); and 8 (16GB, 128GB)

resulting speedups of GMT-Reuse over BaM in Figure 12. In general, as the figure confirms, speedups will increase since there is scope for a larger working set to be accommodated in Tier-2. Further, the benefits are more for applications that have a higher Tier-2 bias than the others, as is to be expected.

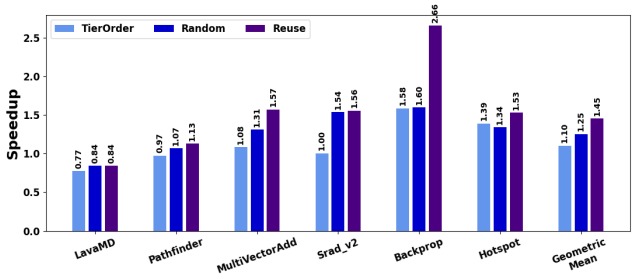


Figure 13. Speedup over BaM for Oversubscription Factor of 2 (Tier-1=32GB, Tier-2=128GB) for Non-Graph Applications

Larger dataset and Tier-1 capacity: Since the statistics for obtaining Figure 7, which has been used to explain the above performance results, involves considerable overheads, the prior experiments used only 16GB of the total 40GB Tier-1 capacity on the A100 GPU. We also conduct experiments with a 32GB Tier-1 capacity, and accordingly increase the application dataset sizes for an over-subscription factor of 2 in Figure 13. We see that GMT-Reuse continues to deliver a 45% speedup compared to the baseline (beating GMT-Random and GMT-TierOrder, by 20% and 35%, respectively), reiterating the importance of **GMT** and our proposed placement/replacement strategy.

3.6 Need for "GPU-Orchestrated" Hierarchy

Our design and implementation of **GMT** adheres to the evolving trends of GPU directly initiating transfers (as in BaM). Placement/replacement/movement of data between Tier-1 and Tier-2 is initiated and performed directly by the GPU. There have been recent works (e.g. [5, 31]) which have also

built a 3-tier hierarchy, albeit using host cores and their operating system (page cache), rather than the "GPU-direct" approach. Prior work has compared BaM with [31], and shown that the GPU-orchestrated throughput-optimized BaM is a much better alternative than the latter despite its 3-tier exploitation of host memory. For completeness, we have also evaluated the recent Heterogeneous Memory Management (HMM) capability that has been added to UVM, to extend it to a 3-tier memory hierarchy (albeit through the host cores and their OS) and compared its performance (using cgroup [1] to limit memory usage) with BaM on our platform and applications. As can be seen in Figure 14, BaM still outperforms HMM - implying that a GPU-orchestrated transfer is much more critical than a CPU-intervened approach despite the latter's Tier-2 leverage. Our `GMT-Reuse` benefits from both - the GPU-orchestrated transfer and Tier-2 host memory - to provide much better performance than each of these individual alternatives (50% over BaM and 357% over HMM) One many wonder which of these 2 factors is more influential in the better performance of `GMT-Reuse` over HMM. To study this, we have further investigated an "optimistic" execution of HMM where its hit rates are made the same as `GMT-Reuse` and its I/O times are accordingly lowered. This is highly optimistic for HMM since a lot of this I/O time may have already overlapped with computation and not fully experienced in its execution. Though not explicitly shown in the interest of space, we find that `GMT-Reuse` still outperforms HMM by 90.3% on the average, even if we give the latter this optimistic benefit. This reiterates the continuing need for GPU orchestrated high throughput transfers over and beyond the much higher hit rates in `GMT-Reuse`.

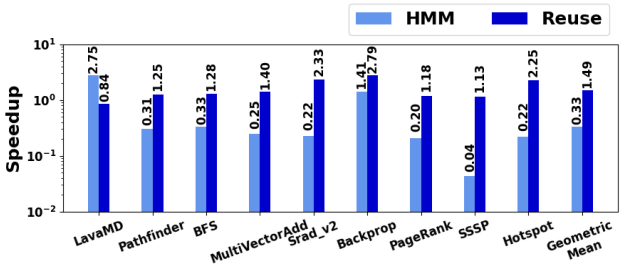


Figure 14. Speedup of HMM and `GMT-Reuse` over BaM

4 Related Work

We have quantitatively compared **GMT** with two - BaM [40] and HMM [5] - recent, representative and most relevant memory tiering mechanisms for GPUs. Below we elaborate on other works, that are indirectly related to our work.

4.1 Extending GPU reach to Storage

CPU-orchestrated data movement: GPUfs [45], ActivePointers [44], GPUDirect Storage [2], Dragon [31] and HMM [5] fall in this category, differing in interfaces to access storage. GPUfs offers a file system interface, similar to POSIX. ActivePointers builds on GPUfs, enabling a mmap interface to avoid explicit I/O. GPUDirect Storage [2] implements a direct data path between GPU and storage, with GPU programs calling a software library explicitly to avail of this feature.

Another set of contributions in this category - Dragon [31] and HMM [5] - extend the virtual memory (i.e. UVM on Nvidia) of the GPUs beyond the (GPU+host) memory to also include storage, by leveraging the page cache of the host operating system. Of these, we have picked Nvidia's implementation of HMM for explicit comparison in this paper, and shown that **GMT** does 357% better than HMM.

G10 [52] is a recent effort that leverages UVM-like (CPU-orchestrated) systems to manage data transfers for deep learning workloads. Through static analysis, it applies compiler-inserted mechanisms to facilitate data transfers between GPU memory, CPU memory, or SSDs in the background. However, this works for such restricted scenarios with regular statically analyzable applications, while **GMT** is a generic runtime system capable of handling various applications, including those with dynamic access patterns (e.g. graphs). Further, unlike a simulation-based evaluation in [52], **GMT** has been implemented on an actual platform which brings out the nuances in orchestrating and tuning the data transfers as shown in this work.

All these prior works rely on the host CPU to initiate and perform the data transfers, which has been noted [40] to become a severe bottleneck towards meeting the throughput needs of numerous cores of the GPU.

GPU-orchestrated data movement: Recent works such as libnvm [32, 33] and BaM [40] completely circumvent the CPU, with the GPU directly initiating and orchestrating the data movement between its memory and storage. This is possible due to GPUDirect RDMA and the NVMe protocol, and is leveraged by **GMT** as well when a evicted page from Tier-1 needs to be directly put into Tier-3, or vice-versa. Additionally, in **GMT**, the GPU also performs the data transfers between its memory and host memory (Tier-2).

4.2 Reuse Distance Based Placement/Replacement

Decades of work on replacement policies have tried to approximate Belady's OPT. While simple LRU approximations (e.g. clock) have typically sufficed in operating systems for large storage capacities, there have been more attempts to predict reuse distances for much smaller structures, e.g. CPU caches [21, 24, 43, 48, 49]. To our knowledge, this is the first work to propose an adaptation for memory tiering in GPUs.

4.3 Tiered Memory Management

Traditionally, operating systems have needed to manage only 2 tiers (main memory and storage). However, technology trends (e.g. non-volatile memories, heterogeneous memories, etc.) has necessitated extending the number of tiers that an

operating system has to manage, e.g. [4, 6, 28, 34, 41, 51]. Many of the issues are quite different in these systems compared to our context and need - no asymmetricies between reads and writes (which many NVM technologies exhibit), capacity ratios, need for GPU orchestration requiring full use of parallelism, need for high throughput transfers by GPU cores rather than by a serial offload engine (DMA), etc.

5 Concluding Remarks & Future Work

This paper has introduced the design and implementation of **GMT**, and shown its considerable benefits over the state-of-the-art by introducing a GPU orchestrated 2nd tier (host memory between GPU memory and storage) when extending the reach of a GPU to capacity needs that only tertiary storage (SSD) can sustain. The contributions of this work are multi-fold: (i) Need for GPU orchestrated management of this 2nd tier, rather than offloading this functionality to software running on the host as is the case with some implementations today; (ii) Need for discretionary placement in this tier rather than simply place the victim from the higher tier; and (iii) Need for carefully tuned mechanisms for data transfers between the top 2 tiers to ensure effective utilization of the GPU cores performing the transfer while maximizing bandwidth. We have designed mechanisms to meet these needs in **GMT**, and demonstrated their benefits with an actual implementation using a diverse set of applications.

There are interesting directions for future work on both hardware and software fronts to enhance **GMT**. On the hardware front, mechanisms for calculating VTD, and support for predicting tiers can help reduce software overheads, to further **GMT**'s benefits. On the software side, asynchronous mechanisms to perform these GPU orchestrations can help reduce the associated costs upon demand misses by performing some of these operations in the background.

6 Acknowledgments

We express special thanks to the anonymous reviewers for their helpful feedback and suggestions. This research has been funded in part by NSF grants 2211018, 1909004 and 1714389.

A Artifact Appendix

A.1 Abstract

We provide the source code as an artifact for **GMT**.

A.2 Artifact check-list

- **Program: C++/CUDA**
- **Compilation: CMake 3.22.1 and CUDA 11.7**
- **Hardware: Nvidia A100 GPU and Samsung 970 EVO Plus SSD**

A.3 Description

A.3.1 How to access. GMT source code can be obtained from https://doi.org/10.5281/zenodo.10873493.

A.3.2 Hardware dependencies. Nvidia A100 GPU and Samsung 970 EVO Plus SSD.

A.3.3 Software dependencies. This requires CUDA 11.7, Linux kernel v5.15.0 with Transparent Hugepages (THP) enabled and Nvidia driver v515.43.04.

A.4 Installation

Please refer to BaM [40] for installation details.

References

[1] Control groups. https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html.
[2] Gpudirect storage: A direct path between storage and gpu memory. https://developer.nvidia.com/blog/gpudirect-storage.
[3] Heterogeneous memory management. https://www.kernel.org/doc/html/v5.0/vm/hmm.html.
[4] Numa balancing. https://mirrors.edge.kernel.org.
[5] Simplifying gpu application development with heterogeneous memory management. https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management/.
[6] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 631–644, New York, NY, USA, 2017. Association for Computing Machinery.
[7] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite, 2017.
[8] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
[9] Chia-Hao Chang, Adithya Kumar, and Anand Sivasubramaniam. To move or not to move? page migration for irregular applications in over-subscribed gpu memory systems with dynamap. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, SYSTOR '21, New York, NY, USA, 2021. Association for Computing Machinery.
[10] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 293–304, 2012.
[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, page 44–54, USA, 2009. IEEE Computer Society.
[12] Nikolay Sakharnykh Chirayu Garg. Improving gpu memory oversubscription performance. https://developer.nvidia.com/blog/improving-gpu-memory-oversubscription-performance/.
[13] Huimin Cui, Qing Yi, Jingling Xue, Lei Wang, Yang Yang, and Xiaobing Feng. A highly parallel reuse distance analysis algorithm on gpus. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 1080–1092, 2012.
[14] Subhasis Das, Tor M. Aamodt, and William J. Dally. Reuse distance-based probabilistic cache replacement. *ACM Trans. Archit. Code Optim.*, 12(4), oct 2015.

[15] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[16] Chen Ding and Yutao Zhong. Reuse distance analysis. Technical Report UR-CS-TR-741, 2001.

[17] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not.*, 38(5):245–257, may 2003.

[18] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 389–400, 2012.

[19] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 389–400, 2012.

[20] Haakon Dybdahl and Per Stenström. Enhancing last-level cache performance by block bypassing and early miss determination. In Chris R. Jesshope and Colin Egan, editors, *Advances in Computer Systems Architecture, 11th Asia-Pacific Conference, ACSAC 2006, Shanghai, China, September 6-8, 2006, Proceedings*, volume 4186 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2006.

[21] Priyank Faldu and Boris Grot. Leeway: Addressing variability in dead-block prediction for last-level caches. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 180–193, 2017.

[22] Hongliang Gao and Chris Wilkerson. A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing. In Joel Emer, editor, *JWAC 2010 - 1st JILP Worshop on Computer Architecture Competitions: cache replacement Championship*, Saint Malo, France, June 2010.

[23] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. Bypass and insertion algorithms for exclusive last-level caches. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 81–92, 2011.

[24] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). *SIGARCH Comput. Archit. News*, 38(3):60–71, jun 2010.

[25] Daniel A. Jiménez and Elvira Teran. Multiperspective reuse prediction. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 436–448, 2017.

[26] T.L. Johnson, D.A. Connors, M.C. Merten, and W.-M.W. Hwu. Run-time cache bypassing. *IEEE Transactions on Computers*, 48(12):1338–1354, 1999.

[27] T.L. Johnson and Wen mei W. Hwu. Run-time adaptive cache hierarchy via reference analysis. In *Conference Proceedings. The 24th Annual International Symposium on Computer Architecture*, pages 315–326, 1997.

[28] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. *SIGARCH Comput. Archit. News*, 45(2):521–534, jun 2017.

[29] M. Kharbutli and Y. Solihin. Counter-based cache replacement algorithms. In *2005 International Conference on Computer Design*, pages 61–68, 2005.

[30] Vikram Sharma Mailthody. *Application Support And Adaptation For High-throughput Accelerator Orchestrated Fine-grain Storage Access*. PhD thesis, University of Illinois Urbana-Champaign, 2022.

[31] Pak Markthub, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Satoshi Matsuoka. Dragon: Breaking gpu memory capacity limits with direct nvm access. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 414–426, 2018.

[32] Jonas Markusse. libnvm: An api for building userspace nvme drivers and storage applications. https://github.com/enfiskutensykkel/ssd-gpu-dma.

[33] Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stensland, and Carsten Griwodz. Smartio: Zero-overhead device sharing through pcie networking. *ACM Transactions on Computer Systems*, 38(1–2), jul 2021.

[34] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[35] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus. *Proc. VLDB Endow.*, 14(2):114–127, oct 2020.

[36] N. Unnikrishnan Nair, P.G. Sankaran, and N. Balakrishnan. Chapter 3 - discrete lifetime models. In N. Unnikrishnan Nair, P.G. Sankaran, and N. Balakrishnan, editors, *Reliability Modelling and Analysis in Discrete Time*, pages 107–173. Academic Press, Boston, 2018.

[37] Victor F. Nicola, Asit Dan, and Daniel M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. *SIGMETRICS Perform. Eval. Rev.*, 20(1):35–46, jun 1992.

[38] Pavlos Petoumenos, Georgios Keramidas, and Stefanos Kaxiras. Instruction-based reuse-distance prediction for effective cache management. In *Proceedings of the 9th International Conference on Systems, Architectures, Modeling and Simulation*, SAMOS'09, page 49–58. IEEE Press, 2009.

[39] Zaid Qureshi. *Infrastructure to Enable and Exploit GPU Orchestrated High-Throughput Storage Access on GPUs*. PhD thesis, University of Illinois Urbana-Champaign, 2022.

[40] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. Gpu-initiated on-demand high-throughput storage access in the bam system architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 325–339, New York, NY, USA, 2023. Association for Computing Machinery.

[41] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Muhammad Aditya Sasongko, Milind Chabbi, Mandana Bagheri Marzijarani, and Didem Unat. Reusetracker: Fast yet accurate multicore reuse distance analyzer. *ACM Trans. Archit. Code Optim.*, 19(1), dec 2021.

[43] Ishan Shah, Akanksha Jain, and Calvin Lin. Effective mimicry of belady's min policy. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 558–572, 2022.

[44] Sagi Shahar, Shai Bergman, and Mark Silberstein. Activepointers: A case for software address translation on gpus. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 596–608. IEEE Press, 2016.

[45] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. Gpufs: Integrating a file system with gpus. *ACM Trans. Comput. Syst.*, 32(1), feb 2014.

[46] E.S. Tam, J.A. Rivers, V. Srinivasan, G.S. Tyson, and E.S. Davidson. Active management of data caches by exploiting reuse information. *IEEE Transactions on Computers*, 48(11):1244–1259, 1999.

[47] Qingsen Wang, Xu Liu, and Milind Chabbi. Featherlight reuse-distance measurement. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 440–453, 2019.

[48] Yunjin Wang, Chia-Hao Chang, Anand Sivasubramaniam, and Niranjan Soundararajan. Acic: Admission-controlled instruction cache. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 165–178, 2023.

[49] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 430–441, 2011.

[50] Lingxiang Xiang, Tianzhou Chen, Qingsong Shi, and Wei Hu. Less reused filter: Improving l2 cache performance via filtering less reused lines. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, page 68–79, New York, NY, USA, 2009. Association for Computing Machinery.

[51] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 331–345, New York, NY, USA, 2019. Association for Computing Machinery.

[52] Haoyang Zhang, Yirui Zhou, Yuqi Xue, Yiqi Liu, and Jian Huang. G10: Enabling an efficient unified gpu memory and storage architecture with smart tensor migrations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 395–410, New York, NY, USA, 2023. Association for Computing Machinery.