Characterizing Emerging Page Replacement Policies for Memory-Intensive Applications

Michael Wu
Computer Science
Yale University
New Haven, USA
mw976@yale.edu

Sibren Isaacman
Computer Science
Loyola University Maryland
Baltimore, USA
snisaacman@loyola.edu

Abhishek Bhattacharjee
Computer Science
Yale University
New Haven, USA
abhishek@cs.yale.edu

***Abstract*—For decades, page replacement in operating systems has referred to the process of moving pages between main memory and disk. During this time, most operating systems used the Clock LRU algorithm for replacement. However, the increased tiering of memory systems has led to the development of new paging algorithms to manage data movement between various memory technologies, algorithms that have often co-opted methods from Clock LRU for page migration. At the same time, the Linux kernel has adopted a new Multi-Generational LRU (MG-LRU) algorithm for page replacement. As memory footprints and hierarchies grow, it is important to understand the key attributes of paging algorithms that determine their performance on various workloads and system configurations.**

This work presents the first characterization of multiple MG-LRU configurations on various memory-intensive workloads for SSD and ZRAM swap. Our experiments show that MG-LRU exhibits high performance variation across otherwise identical workload executions. We also show that the relative performance of MG-LRU compared to Clock LRU is highly variable across different configurations of the surrounding system. Finally, we confirm that simple adjustments to MG-LRU parameters are not a panacea to these issues. Broadly, our work illuminates the complex relationship between workloads, system configurations, and replacement policies and motivates further work to profile, re-invent, and ultimately optimize memory management in computer systems.

I. Introduction

Paging algorithms are as old as operating systems themselves. For decades, the most common paging algorithm in commercial systems was a form of the Clock LRU approximation algorithm used by the Linux kernel [2], [10]. Traditionally, paging has referred to the process of migrating memory pages solely between DRAM and disk, and becomes relevant only when the combined memory usage of running applications approaches or exceeds the system's memory capacity. Recently, however, the emergence of multi-tiered memory systems has motivated the development of new policies orchestrating page movement not just between DRAM and disk, but also between various memory technologies [7], [23], [26], nodes [19], [22], and accelerators [6], [11], [20]. This has sparked a renewed academic interest in understanding the mechanisms used by computer systems to profile, classify, and migrate pages based on their access patterns.

While a number of page migration policies have been proposed in academic literature, the Linux kernel has adopted a new paging algorithm as well. This algorithm, known as multi-generational LRU (MG-LRU), claims improvements in both paging decision quality and profiling overheads over the previous Clock algorithm [4], [14], [15]. At over one billion users [3], MG-LRU has become the most prevalent paging algorithm in commercial systems. Yet, while MG-LRU leverages a number of new data structures, such as bloom filters and PID controllers, to manage page tracking and movement, no literature exists examining the contributions of each design decision on paging performance. As memory footprints continue to grow and memory hierarchies become increasingly tiered, understanding these structures — and their deeper insights into page access tracking — becomes crucial.

In this paper, we present the first characterization of MG-LRU on a variety of memory-intensive applications. Our evaluation uses benchmarks from three different domains: data warehousing, graph processing, and key-value stores — making them representative of common datacenter workloads. We perform an extensive grid search of configurations, testing the effects of tuning various MG-LRU parameters as well as how the interact with different swap mediums. Our results show that MG-LRU performance can be remarkably inconsistent depending on the workload, memory usage-capacity ratio, and swap medium. Specifically, we show the following:

- Under high memory pressure, while MG-LRU successfully improves average performance compared to Clock LRU, it can significantly increase runtime variation in some cases. These differences are attributable to a decrease in the average page faults per workload execution, but a significant increase in their variance.
- We show that no single configuration of MG-LRU performs best on every workload-system combination. In some cases, the best-performing configuration in one setting becomes the worst-performing in another.
- We find that with a faster swap medium, the number of page faults per execution consistently increases. This suggests that reducing overhead of page access tracking relative to the cost of swapping is crucial for improving the quality of replacement decisions.

Overall, this work is an early characterization that points towards future areas of study and key issues that future

algorithms must consider. In particular, our work suggests that there is no "one size fits all" paging algorithm that minimizes both mean runtime and runtime variance for all workloads and system configurations. Instead, further study is required to identify all major sources of variance in the system, and to develop new algorithms that are robust to varying workloads and configurations.

II. Background and Related Work

A. Page Access Tracking

Fundamentally, the goal of a replacement policy is to determine whether any page is hot (likely to be accessed) or cold (unlikely to be accessed). Hot pages will be kept in main memory, while cold pages can be safely swapped out. Replacement policies commonly exploit *Accessed* bits stored in page table entries (PTEs) to distinguish hot and cold pages. These bits are set whenever the PTE is the target of a hardware page walk [27]. Current replacement policies track access patterns by periodically scanning these bits in the page table. When the system encounters a set accessed bit, it can infer that the page was accessed relatively recently, guiding it to increase its estimation of the page's hotness. The accessed bit is then reset to allow the hardware to provide information on future accesses.

Some page migration policies have also used page poisoning to obtain information on page accesses. This purposely marks a page's PTE as inaccessible (e.g., by marking it as not present or reserved), triggering a fault on the next access to that page [7], [13]. This method obtains precise information on the accessed pages and their exact access time. However, the cost of page faults to frequently accessed pages can become prohibitively expensive. As such, most page poisoning approaches combine it with sampling methods.

B. Clock-LRU

For decades, the Linux kernel used Clock-LRU as its page replacement policy. This algorithm, also known as Clock, LRU Second Chance, and 2Q, approximates LRU using two queues: the *active* and *inactive lists* [2]. The goal of the Clock algorithm is to ensure the active list contains the working set of all running processes while the inactive list contains candidates for page eviction/reclaim. This is done by periodically scanning the accessed bits of pages at the bottom of the active list. If a page has not been accessed, it is moved to the inactive list. Otherwise, it is placed at the top of the active list. At reclaim time, the kernel scans the accessed bits of pages on the inactive list. If a page has been accessed, it is moved to the active list. Otherwise, it is reclaimed or evicted from memory.

C. Page Migration Policies

Emerging systems have begun using multiple memory tiers to improve memory-intensive applications' performance while optimizing the total cost of ownership. This has motivated research to uncover the best method for dynamically migrating pages between tiers. By default, Linux uses AutoNUMA for managing memory tiers [13] AutoNUMA aims to match processes and the memory pages they access to the same node. However, because it was not designed to support CPU-less memory nodes, it lacks mechanisms to demote pages, limiting its performance in contexts with memory tiering [21]. Thermostat [7] and MTM [26] sample access information using page poisoning and access bit scans, respectively. Both define a notion of page hotness based on access recency and frequency, promoting or demoting pages based on (possibly dynamic) hotness thresholds. TPP [23], on the other hand, is directly built on top of the data structures used for Clock. It adapts Clock for page migration by having evictions to target lower memory tiers instead of disk. It then promotes accessed pages depending on their presence in the active or inactive list. These works indicate a large design space exists for page migration algorithms. Yet, all approaches fundamentally rely on classifying hotter and colder pages to guide migration decisions.

III. MG-LRU Overview

MG-LRU claims to improve over Clock in both the accuracy of replacement decisions and the efficiency of page access tracking. It does so by introducing an alternate method for tracking page accesses and using multiple generation lists to classify page hotness.

A. Generations and Aging

MG-LRU replaces the active and inactive lists with multiple generation lists. Broadly, as the kernel tracks page accesses, idle pages will move towards older generations, while accessed pages are moved to the youngest generation. This allows a more precise classification of page hotness across a spectrum of generations rather than simply active or inactive. In theory, this scheme also results in fewer pages being accessed after reaching the oldest generation, allowing page reclaim to proceed more aggressively [15]. We examine the effect of generation count in §V-B.

B. Page Access Tracking and Filtering

When Clock scans accessed bits, it iterates through the physical frame numbers of pages in the active and inactive lists. Each scan triggers a physical-to-virtual address translation to access the page's PTE, which requires walking the reverse map, a pointer-based data structure that is expensive to access [15], [24]. To alleviate this, MG-LRU introduces an aging thread that scans page tables linearly. Doing so avoids the cost of physical-to-virtual translations and also takes advantage of spatial locality in the page table itself. However, naive linear scans of the page table are wasteful when there are many mapped but unallocated regions in the virtual address space. As such, MG-LRU uses a bloom filter to limit scanning to regions of the page table that are likely to map frequently accessed pages [4]. By default, MG-LRU adds a region of the page table to the bloom filter if it contains at least one accessed PTE per cache line. The bloom filter is then used to filter scans of the next generation. We examine the effectiveness of this data structure in §V-B.

C. Eviction

In addition to the aging thread, MG-LRU maintains an eviction thread that behaves similarly to Clock. At reclaim time, the eviction thread scans the oldest generation for reclaim candidates. Like Clock, the eviction thread walks the reverse map before marking a page for eviction. If the page has been accessed, it is promoted to the youngest generation instead. Unlike Clock, the eviction thread will also scan the surrounding PTEs of any accessed pages it detects through the reverse map. This again takes advantage of spatial locality to reduce the overhead of page table scans. It also creates a feedback loop between the aging and eviction threads, as regions scanned by the eviction thread may be added to the bloom filter for later aging scans.

D. PID Control

MG-LRU makes an exception to its promotion and eviction rules for pages accessed via file descriptors. Because such pages are often accessed only once (e.g., reading input data from a file), MG-LRU does not promote them to the youngest generation. Instead, such pages are promoted by a single "tier" within a given generation. This avoids cases where such sparsely accessed pages are promoted above hot pages. Still, it can degrade performance if file descriptor pages are accessed frequently (e.g., when an application performs significant buffered I/O). To manage this, MG-LRU tracks how often pages in each tier are accessed soon after eviction, or "refaulted." If the refault rates in higher tiers — which contain only pages behind file descriptors — are higher than that of the lowest tier of a generation, MG-LRU will protect higher tiers from eviction until the rates are balanced. This process is managed by a proportional-integral-derivative (PID) controller [4], [14]. Since our tested workloads do not perform significant file descriptor accesses, we do not test the effects of tuning the PID controller, leaving it instead for future work with workloads affected by it.

IV. Methodology

To characterize MG-LRU, we use workloads from three domains: data warehousing, graph processing, and in-memory key-value stores. We select these domains to be representative of common datacenter workloads. Specifically, we run TPC-H [1] using Spark-SQL [8], PageRank from the GAP Benchmark Suite [9], and YCSB A, B, and C [12] using Memcached [18]. All tested workloads have a memory footprint between 12-16GB. Although Spark has mechanisms to spill data to disk when its memory is oversubscribed, we configure its memory limits to avoid spilling to match its evaluation in memory tiering work [28]. To characterize performance variation for TPC-H and PageRank, we run 25 executions of both workloads in each tested system configuration. To avoid confounding effects from memory fragmentation, we reboot the system before each execution. For YCSB, we collect tail latency statistics on 110 million requests after initially loading 11 million items to the cache.

We test Clock, MG-LRU, and variants of MG-LRU on the Linux kernel version 6.8. We perform our experiments on a machine using an Intel Core i7-8700 CPU at 3.2GHz with 6 cores and 12 threads. We run Spark and PageRank with 12 threads and Memcached with the default four threads. In our experiments with SSD swap, we measure the latency of 4KB reads and writes to be ~ 7.5ms. We also perform experiments with ZRAM swap, which reserves a section of memory to store compressed data [5]. This allows a system to slightly increase memory capacity at a lower cost than swapping to disk. Because of its speed, ZRAM swap approximates the effects of slower memory tiers, such as remote or disaggregated memory [19], [22], [28]. We configure ZRAM to use LZO-RLE compression and measure the 4KB read and write latencies be 20μs and 35μs, respectively.

V. Characterizing MG-LRU

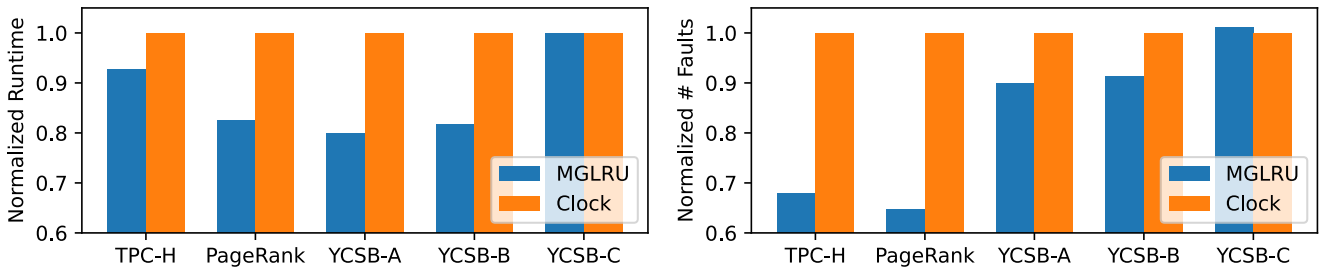
We begin our characterization by comparing MG-LRU with its Clock-LRU predecessor on systems using SSD swap.

A. Revealing Performance Variation

Figure 1a shows the average total execution time of each benchmark with both replacement policies. We see that under high memory pressure, MG-LRU matches or outperforms Clock on all benchmarks in average performance. Figure 1b shows that this improvement arises from decreased swapping when using MG-LRU. This indicates that, in general, multiple generations and page table scanning successfully improve the quality of page replacement decisions. However, this does not reveal the whole story.

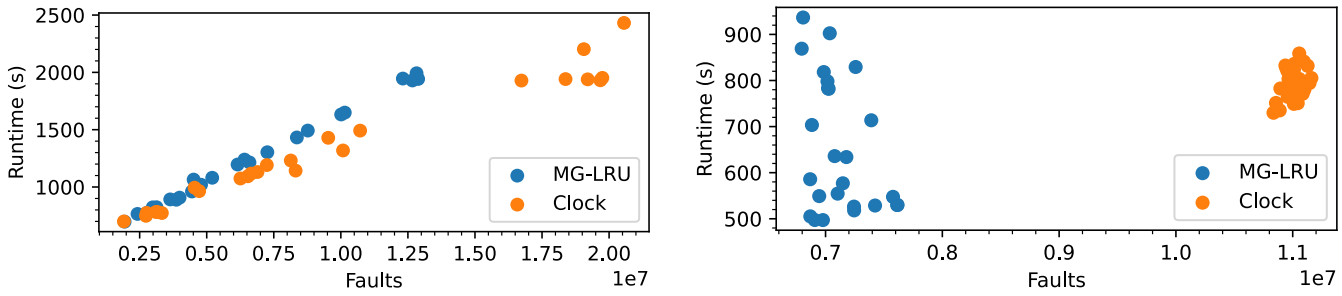
Although MG-LRU consistently improves average performance, this is not the case for performance variation. Figure 2 shows the joint distributions of execution times and faults across all trials of the TPC-H and PageRank workloads. For TPC-H, we see that MG-LRU successfully reduces performance variation compared to Clock. However, the execution times still range from $\sim$700 seconds to over 2000 seconds, nearly a $3\times$ increase between the fastest and slowest execution. This performance variation is seen again for PageRank, where the factor between the fastest and slowest executions is nearly $2\times$. In contrast with the previous benchmark, on PageRank, MG-LRU significantly increases performance variation over Clock, which itself performs consistently. In fact, the variation with MG-LRU is so high that, although MG-LRU reduces average performance by over 15%, the worst-case execution time is *higher* than that of Clock. This has large implications on the use of MG-LRU in a datacenter environment, as it is well-known that such stragglers are more critical to performance than the nodes with the fastest executions [16], [31].

Another striking difference between performance on TPC-H and PageRank is the relationship between the number of page faults and overall execution time. For TPC-H, we see a nearly perfect linear relationship between faults and execution time. In fact, for all experiments with this system configuration, we see a coefficient of determination (r^2) of



(a) Execution Time **(b) Faults**

Fig. 1: Average execution time and total fault counts normalized to Clock-LRU. MG-LRU consistently improves performance over Clock in this system configuration. Experiments use SSD swap and a 50% memory capacity-to-footprint ratio. For YCSB workloads, we normalize the average request time over the workload's execution.



(a) TPC-H **(b) PageRank**

Fig. 2: Joint distributions of execution time and faults. Both MG-LRU and Clock experience high performance variation on TPC-H. However, Clock has a far tighter runtime distribution on PageRank.

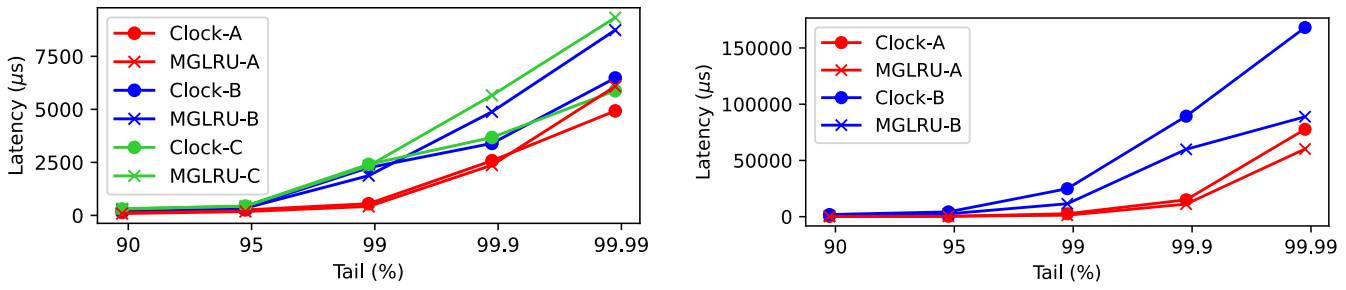
over 0.98 for linear regression. Meanwhile, the opposite is true for PageRank, where the number of faults appears to have no correlation with overall runtime. Intuitively, this would call into question whether the change in replacement policy is the source of variance, as one would expect faults to dominate runtime at high memory pressure. And yet, the difference in distribution spread between MG-LRU and Clock clearly indicates that MG-LRU is a major source of variance, even if it is not through inconsistent fault rates. This leaves the scanning method as the only other potential source of variation, which we explore further in §V-D.

We also examine the effects of using MG-LRU on tail latency. Again, while MG-LRU always achieves equal or greater average throughput than Clock, its benefit becomes less clear when considering tail performance. Figure 3 shows the tail latencies of both read and write requests for the YCSB A, B, and C workloads. Our results show that using MG-LRU trades off higher read latencies for shorter write latencies. For reads, both replacement policies perform similarly up to the 99% latency, after which latencies grow much faster with MG-LRU, approaching a 20-40% increase at the 99.99% tail. Meanwhile, the opposite trend is true for write latencies, with Clock experiencing 10-50% higher latencies after the 99% tail. These trends suggest that deciding between MG-LRU or Clock is not as simple as selecting the policy with the best throughput. Instead, our results indicate that the choice of replacement policy may depend heavily on the application's characteristics, whether it is read- or write-heavy, and what tradeoff between throughput and tail latency is best for its end goal.

B. Tuning MG-LRU Parameters

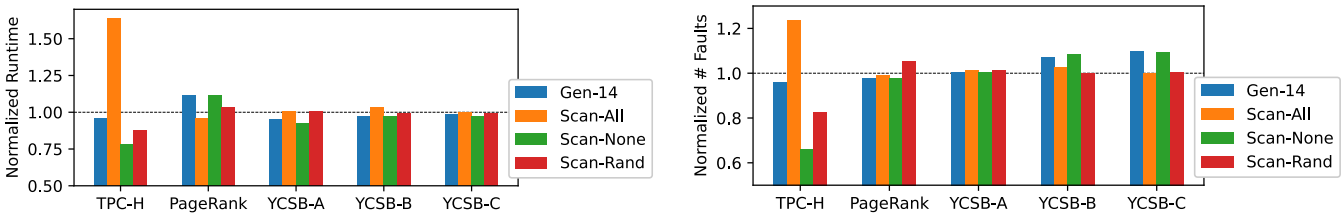
We now examine the effects of adjusting MG-LRU parameters on mean and tail performance. First, we adjust the number of generations lists used to track pages in physical memory. When investigating the motivation for why MG-LRU uses only four generations by default, we found it was simply to double the number of lists used by Clock[1]. Even with this increase, we find many cases where MG-LRU is at the maximum number of generations and, therefore, cannot create a new youngest generation for the next aging scan. This causes multiple consecutive scans to promote pages all to the same generation, which reduces the precision to which a page's generation number implies its access recency. Note that moving page metadata between generation lists is an O(1) operation, so increasing the number of lists adds negligible overhead. As such, we can safely increase the maximum number of generation lists so that MG-LRU can always increment the youngest generation after an aging scan. We find that using 2^{14} generations more than achieves this condition for our workloads and system configurations without

[1]As noted by the developers in the source code.



(a) Read Latencies **(b) Write Latencies**

Fig. 3: Tail latency distributions. MG-LRU significantly improves write tail latencies over Clock, but exhibits longer read tail latencies. Since YCSB-C consists only of read requests, no write tail latencies are shown.



(a) Execution Time **(b) Faults**

Fig. 4: Mean execution time and faults of alternate MG-LRU configurations. Results vary significantly both by the configuration and workload executed, making it unclear what the "best" MG-LRU configuration is.

inducing perceivable overheads compared to lower generation counts. We refer to this configuration of MG-LRU as *Gen-14*.

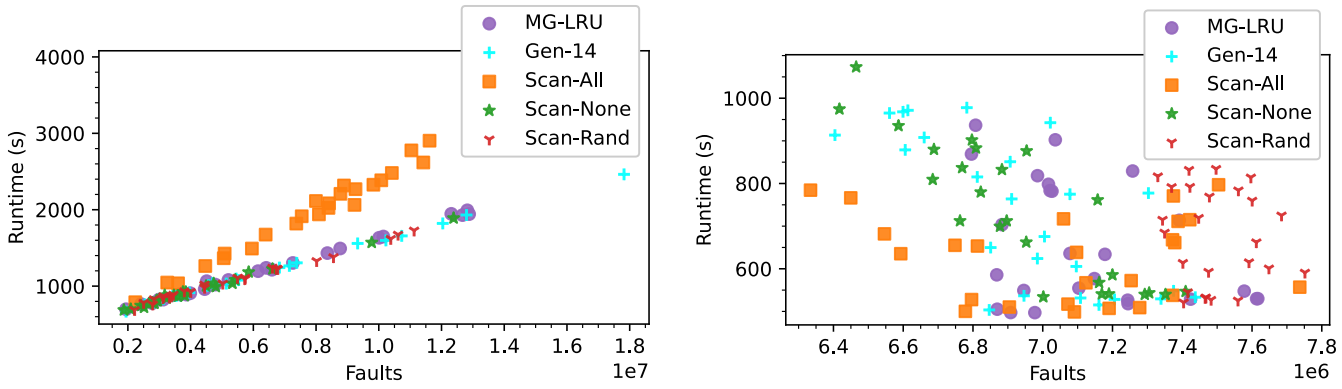
In our testing, increasing the generation count slightly improves average performance across most benchmarks (Fig. 4a). However, this improvement is inconsistent, with *Gen-14* underperforming on PageRank. Moreover, due to the high runtime and fault variance exhibited by both policies, we find no statistically significant differences in average performance ($p > 0.05$). Broadly, this result is to be expected, as the generation count does not change the core promotion and eviction policies in MG-LRU. In both cases, accessed pages are promoted to the head of the youngest generation list, while non-accessed pages are eventually evicted from the tail of the oldest generation list.

After adjusting the generation count, we identify the bloom filter as the next parameter-tuning target. Since the bloom filter identifies the sections of the page table scanned by the aging thread, it is arguably the most important data structure for determining MG-LRU performance. First, we test whether the bloom filter is important for performance. We test three methods for removing the impact of the bloom filter on aging: scanning the entirety of the page table (*Scan-All*), scanning none of the page table (*Scan-None*), and scanning each section with a 50% probability (*Scan-Rand*). The first two configurations effectively disable the bloom filter by not filtering any sections and filtering all sections, respectively, while the third attempts to reduce scanning overheads while still probabilistically scanning the entire page table.

Figure 4 shows mean performance results with the same capacity and swap configuration as §V-A, normalized to the performance of the default MG-LRU configuration. It is immediately clear that MG-LRU performance is also inconsistent with respect to the choice of its parameters. Within TPC-H, an enormous gap exists between *Scan-None*, which improves on MG-LRU by over 20%, and *Scan-All*, which degrades mean performance by over 60%. However, the opposite trend exists for PageRank, albeit not as drastically. *Scan-None* slightly degrades performance, while *Scan-All* — by far the worst configuration on TPC-H — performs best. This indicates that the relative ordering of MG-LRU parameter configurations is also inconsistent across workloads.

All MG-LRU configurations perform similarly on YCSB workloads. This is likely due to the highly skewed zipfian access distribution used to generate requests [12]. It is known that LRU is a suboptimal replacement policy for such distributions, with many software key-value cache applications using a variant of the FIFO eviction policy [17], [29], [30]. Given that MG-LRU attempts to approximate LRU, it is expected that all of its variants are ultimately limited in effectiveness.

Figure 5 allows us to examine the behavior of each MG-LRU configuration in more detail. As before, we see a strong linear relationship between faults and execution time for TPC-H. Interestingly, the slope of this relationship is the same for all parameter configurations except *Scan-All*, which appears to have a higher runtime cost per fault. We find this to result from increased straggler threads. This occurs because *Scan-All* scans many more sections of the page table compared to other MG-LRU variants. Combined with how the aging thread linearly scans the page table, this leads to the scanned pages of each thread's working memory being bimodally distributed. That is, most threads have either had all of their working memory recently scanned, or none of them. This leads



(a) TPC-H **(b) PageRank**

Fig. 5: Joint execution time and fault distributions for TPC-H and PageRank using alternate MG-LRU configurations. TPC-H continues to exhibit a strong linear relationship between runtime and the number of faults. Meanwhile, the runtime of PageRank appears less correlated with the total fault count.

to inconsistencies in how thread-specific pages are evicted, causing stragglers even when all threads execute the same type of task. In contrast, all other tested MG-LRU variants have mechanisms to have more consistent scanning behavior across all threads, whether by the bloom filter (*MG-LRU* and *Gen-14*), use of randomness (*Scan-Rand*), or not scanning at all (*Scan-None*). Note that this highlights scanning overhead as a critical factor for paging performance. Such inconsistencies would become less likely to occur as scans become faster with respect to the application.

Moving forward, Figure 5a shows how *Scan-None* performs best on TPC-H — it has the lowest mean and spread of faults. This seems unintuitive, as *Scan-None* only scans accessed bits through the eviction thread. Thus, it is the most similar to Clock among our tested configurations, yet it improves performance on TPC-H instead of degrading it. However, the critical difference between *Scan-None* and Clock is that Clock walks the reverse map for every physical page individually, incurring the cost of pointer chasing each time. Meanwhile, when *Scan-None* finds an accessed page table entry through the reverse map, it takes advantage of spatial locality by also scanning the surrounding page table entries. This can significantly reduce the cost of page table scans which, as shown by *Scan-All*, is critical for TPC-H.

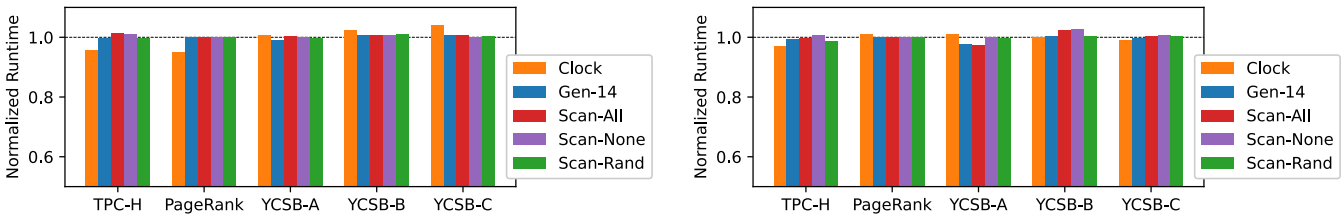
We see this locality benefiting on PageRank as well, where *Scan-None* again outperforms Clock. In general, though, PageRank seems much less sensitive to scanning overheads than TPC-H. This is demonstrated by *Scan-All*, which is by far the worst policy for TPC-H, but the best for PageRank. While more study is ultimately needed, we suspect this results from the different threading models in each workload. Since TPC-H is executed using Spark-SQL, it is split into a number of highly parallel stages with little synchronization overhead and mostly balanced work per thread. Meanwhile, PageRank consists of multiple iterations of parallelized sparse matrix multiplication. This makes the work per thread vary with the degree of each graph vertex. Thus, if the end of an iteration is spent waiting for the processing of a single high-degree vertex, the overall runtime can be affected more by a few critical faults rather than the overall fault rate. Additionally, such scenarios would experience less CPU contention, allowing expensive page table scans to improve the accuracy of critical replacement decisions while incurring little cost on overall runtime.

Overall, tuning MG-LRU parameters reveals a complex relationship that exists between the overhead of scanning accessed bits, the quality of replacement decisions, and the sensitivity of workloads to these factors. Our tested configurations show that this relationship greatly affects the "optimal" parameters for any given scenario. At a higher level, it is remarkable that using these parameter configurations successfully improved both mean performance and variation in any workload. Recall that each of these configurations effectively removed the bloom filter, a major data structure in MG-LRU's design. In the case of *Scan-Rand*, this data structure was replaced with an entirely random method of scanning the page table. This calls into question whether the bloom filter is a necessary data structure for MG-LRU, which we discuss further in §VI-C. In the following sections, we continue to explore the use of MG-LRU in varying configurations of the surrounding system.

C. Trends Across Memory Capacities

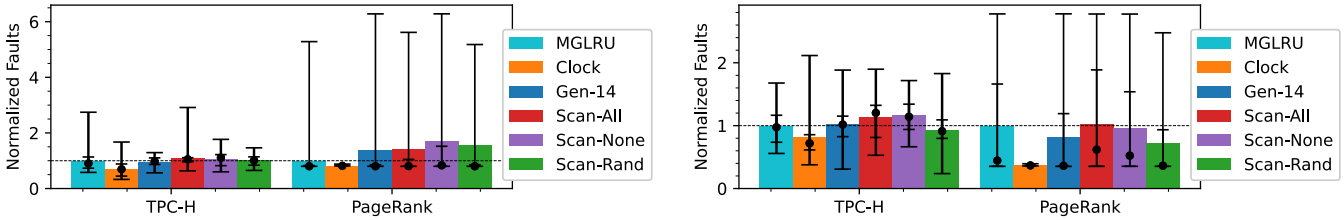
So far, we have characterized MG-LRU in settings where the available memory for each workload was only 50% of the workload's memory footprint. This represents heavy memory pressure, which induces significant swapping and magnifies the replacement policy's performance. We now present results for contexts with less stringent constraints, where 75% and 90% of the workload's memory footprint can fit in main memory.

Figure 6 shows mean performance results for all tested replacement policies, normalized to that of MG-LRU with default parameters. We find that each replacement policy performs within a few percentage points of each other across all workloads. This results from the number of faults decreasing significantly, reducing their impact on overall runtime. We do



(a) 75% Capacity-Footprint Ratio **(b) 90% Capacity-Footprint Ratio**

Fig. 6: Mean performance at varied capacity-footprint ratios. Performance is normalized to the mean performance of MG-LRU using default parameters. Due to the reduced fault rates at higher memory capacities, total application runtime is more balanced across all replacement policies.



(a) 75% Capacity-Footprint Ratio **(b) 90% Capacity-Footprint Ratio**

Fig. 7: Normalized fault distributions at different capacity-footprint ratios. Error bars show maximum, minimum, and all quartiles. Faults are normalized to the mean number of faults when using MG-LRU with default parameters. While runtime variation is lessened at higher memory capacities, variance in the number of faults significantly increases.

see a surprising result between MG-LRU and Clock. While MG-LRU consistently outperformed Clock at a 50% capacity-footprint ratio, we see cases where Clock shows improvements over MG-LRU at higher memory capacity. Although these improvements are relatively small (2-5%), they are statistically significant in all cases ($p < 0.01$).

While the total runtime variation is relatively small in our experiments, the variation in number of faults is not. Figure 7 shows the fault distributions for each policy on TPC-H and PageRank. At a 75% capacity-footprint ratio, we see a large range of faults for every MG-LRU configuration on PageRank. In some cases, there are executions where the number of faults is more than 6× the average. Yet, the interquartile range in each of these cases is often negligible in comparison. This indicates that the range results from a few outlier executions. That being said, the presence of such outlier executions across all MG-LRU configurations raises concerns that MG-LRU may be vulnerable to such corner cases. Further study is required to determine the cause of such corner cases and why they do not occur for Clock, which has a comparatively tight fault distribution on PageRank.

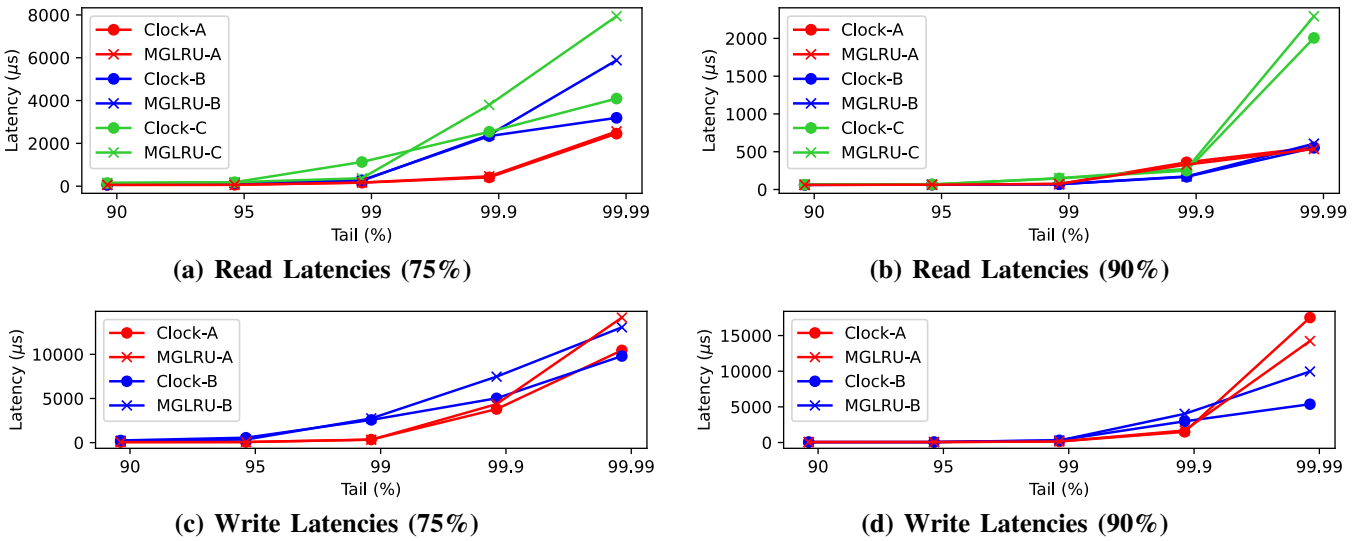
Regarding tail latencies, we see inconsistent trends across varying memory capacities. Figure 8a shows a similar trend seen in §V-A, where Clock exhibits lower read latencies than MG-LRU. However, instead of increasing write latencies as in §V-A, Clock also reduces these latencies at 75% capacity (Fig. 8c). This trend becomes increasingly irregular at 90% capacity, where comparisons of write latencies become workload dependent (Fig 8d). Meanwhile, the read latencies start to converge with increasing memory capacity (Fig 8b), an expected result.

Overall we see that just as MG-LRU's relative performance is inconsistent across workloads, it can also vary with memory capacity and oversubscription ratio. More broadly, these results introduce the concept that different replacement policies may be best suited for specific memory capacities or oversubscription ratios. To our knowledge, this concept has yet to be studied in detail in academic literature. Yet, as memory capacities and footprints continue to grow, the effectiveness of paging policies at scale will only increase in relevance.

D. Varying Swap Medium

For our final set of experiments, we investigate using ZRAM as the swap medium instead of SSD. We configure ZRAM to use the LZO-RLE compression and limit the uncompressed memory capacity to 50% of the workload footprints. With 4KB read and write latencies in the tens of microseconds, our ZRAM configuration is comparable to systems with remote or network-disaggregated memory [19], [22], [28].

Figure 9 shows mean performance results. We once again see performance trends that differ from previously tested system configurations. While the performance across MG-LRU parameters is consistent, Clock now matches MG-LRU performance in all workloads except PageRank. Figure 10 shows that the fault distribution coincides perfectly with these results. It is possible that on PageRank, the combination of lower fault costs, a semi-irregular access pattern, and not taking advantage of locality makes accessed bit scans too slow when using Clock, leading to poor replacement decisions. In contrast, TPC-H has more regular access patterns, allowing



(a) Read Latencies (75%)

(b) Read Latencies (90%)

(c) Write Latencies (75%)

(d) Write Latencies (90%)

Fig. 8: Tail latency distributions at different memory capacities. Just as in experiments at 50% capacity, Clock exhibits lower tail latencies for reads than MG-LRU. However, the write tail latencies are more unstable with changing memory capacity. Because all variants of MG-LRU exhibit similar tail latencies, we only show results for the default parameters.

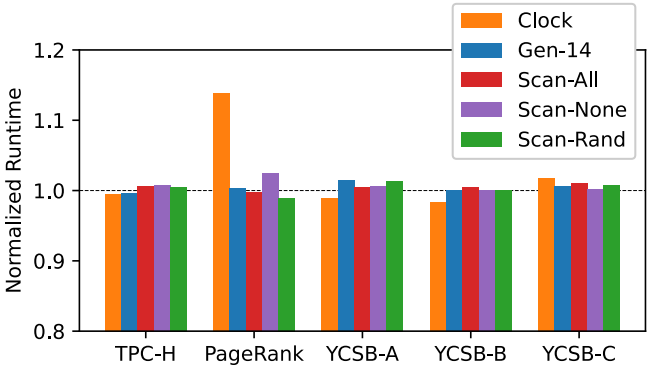


Fig. 9: Mean performance using ZRAM swap at a 50% capacity-footprint ratio. With the exception of PageRank, Clock and MG-LRU perform equally.

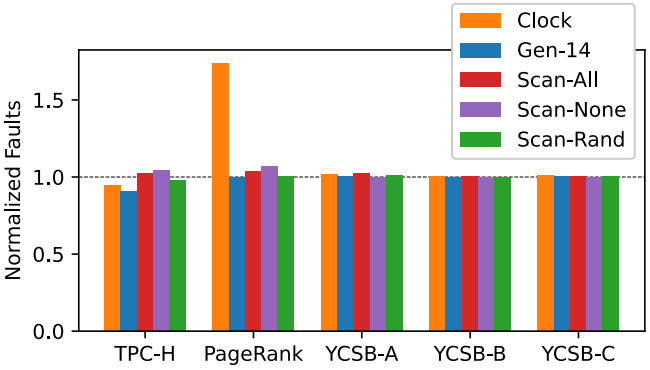


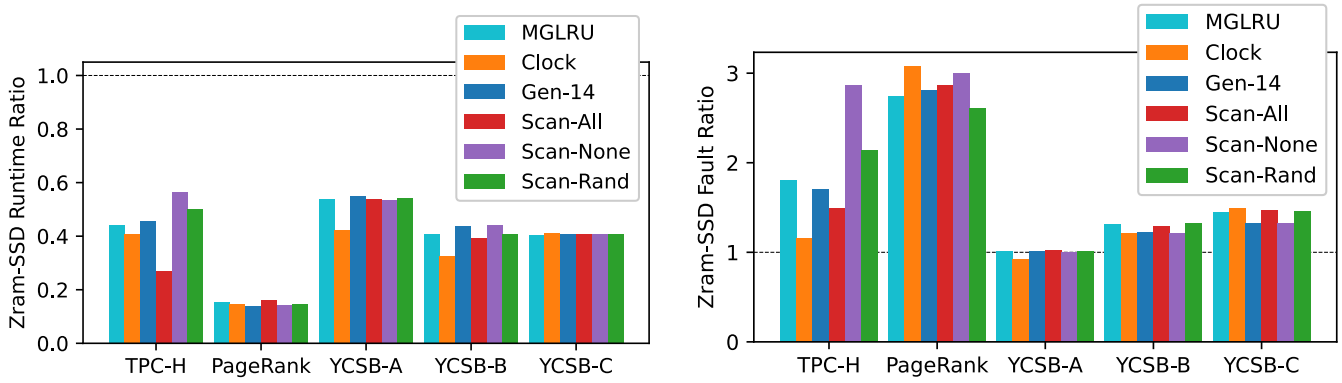
Fig. 10: Mean faults using ZRAM swap at a 50% capacity-footprint ratio. As with mean runtime, Clock faults equally as much as MG-LRU in all workloads except PageRank.

slower scans to still have effect, and YCSB consists of random accesses, limiting the effectiveness of all policies.

To further examine the relationship between the cost of swapping and the quality of replacement decisions, we plot the change in runtime and faults when using ZRAM instead of SSD swap (Fig. 11). As expected, fault rates do not increase much with YCSB workloads due to their random access patterns. For workloads B and C, we likely see slightly increased fault rates because their proportion of write requests is much lower than that of workload A [12]. As a result, A will exhibit more synchronization overheads than B and C, reducing the relative swap-induced overheads, which masks the effect of swap costs on scanning and replacement decisions.

We see that for the workloads with more regular access patterns, TPC-H and PageRank, the MG-LRU faults significantly more using ZRAM instead of SSD swap. This is highlighted the most in PageRank, which runs over 5× faster yet faults nearly 3× more with ZRAM swap. As mentioned before, this indicates that as the cost of swapping decreases, the page table scans do not progress quickly enough to make accurate replacement decisions as the application runs. This bears significance for emerging systems with remote or tiered memory, as the page replacement and migration policies will require faster methods of tracking access information to maintain the same fault rates.

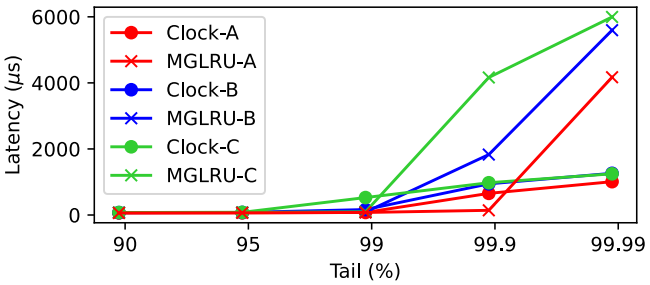
Lastly, we examine the effect of ZRAM swap on the tail latency of YCSB workloads. Figure 12 shows our experiment results. Since all MG-LRU variants exhibit similar tail latencies, we show only the default parameter configuration. Across all workloads, MG-LRU exhibits 2-5× longer 99.99% tail latencies. This differs considerably from experiments using SSD swap, where MG-LRU exhibited longer read tails, but shorter write tails. With ZRAM swap, Clock exhibits slightly higher 99% tails in some cases, but heavily outperforms MG-LRU afterwards. Since Clock also matches MG-LRU in throughput, another difference from SSD swap, our ex-



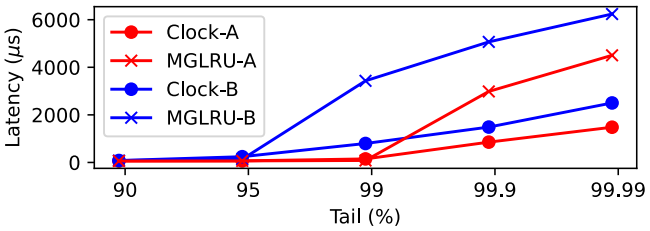
(a) Runtime difference between SSD and ZRAM

(b) Difference in faults between SSD and ZRAM

Fig. 11: Change in runtime and number of faults between ZRAM and SSD swap. Despite the significant decrease in runtime when switching to ZRAM, the number of faults remains steady or increases significantly.



(a) Read Latencies



(b) Write Latencies

Fig. 12: Tail latency distributions using ZRAM swap. Under this configuration, Clock strictly and significantly outperforms MG-LRU in tail performance.

periments suggest that Clock is a strict improvement over MG-LRU in this system configuration. Given that MG-LRU heavily outperforms Clock in other system configurations and workloads, such a result indicates that replacement policies have very complex interactions with different workloads and systems. This calls for deeper study of all aspects of replacement policies, especially as memory footprints, capacities, and hierarchies continue to grow.

VI. Discussion and Future Work

This work characterizes MG-LRU across a number of workloads and system configurations. Doing so highlights a number of tradeoffs to consider in the future characterization and design of page replacement and migration policies.

A. MG-LRU and Clock

When comparing MG-LRU to its predecessor, Clock, we find that MG-LRU is not necessarily an improvement. While MG-LRU improves mean performance considerably in many scenarios, in §V-A and §V-D, we show cases where MG-LRU has considerably worse tail performance than Clock, with MG-LRU performing strictly worse on YCSB with ZRAM swap. Intuitively, one could expect MG-LRU to induce more performance variation due to its use of a separate aging thread to scan page tables. This thread introduces more sources of CPU contention and scheduling variance, which circularly affects the results of each page table scan. Additionally, MG-LRU exploits page table locality in the eviction thread to amortize the cost of reverse map walks. This reduces the average overhead of page table scans. However, compared to Clock, it can also cause the eviction thread to scan far more accessed bits before reclaiming a page. Thus, in the tail case, MG-LRU could experience a far slower rate of page reclaim than Clock. In a state of high memory pressure or thrashing, this could make reclaim too slow to make space for demand faults from the application, forcing page faults to wait for disk writes to complete before being serviced. Such a case could occur if the oldest generation contains many accessed pages from separate regions of the page table, a possible scenario under YCSB's random access distribution and an explanation for MG-LRU's increased tail latencies.

We leave a more detailed investigation of such cases for future work. More broadly, our experiments indicate that choosing whether to use MG-LRU or Clock, especially in a datacenter environment, is not as simple as selecting the policy with the best mean performance. Instead, one must deeply understand the complexities of the executed workloads and how they interact with the underlying system and replacement policy. We hope to motivate research that develops a better understanding from the paging algorithm's point of view, thus reducing the need for application-specific profiling.

B. Scanning Overhead and Quality

Our experiments also reveal insights regarding the effect of scanning overhead on the quality of replacement decisions. This is exemplified in §V-D. We measure the latency of swapping to ZRAM to be two orders of magnitude faster than swapping to SSD. Yet, despite the far faster runtime that results from this, there are cases where using ZRAM leads to many times more faults than SSD. The key difference between the two cases is that when using SSD swap, applications spend more overall runtime waiting for faults, allowing page table scans to progress further before the application continues execution. This indicates that when LRU is a suitable replacement policy for an application, an increase in the speed of scans relative to the speed of the swap medium could directly translate to improved replacement decisions and, ultimately, faster execution. Such a result motivates further research into more efficient scanning methods, possibly including hardware support for setting and scanning accessed bits.

C. Use of Randomness

In examining alternate parameter configurations of MG-LRU, we see unexpected results from *Scan-Rand*. Instead of using the bloom filter to reduce wasteful page table scanning, *Scan-Rand* removes this data structure entirely, iterating over each region of the page table and scanning it with a naively chosen 50% probability. Yet, we find that *Scan-Rand* does not degrade performance compared to the default MG-LRU configuration. In fact, it improves performance in some cases. This raises questions about the bloom filter's utility and whether it is a useful data structure in MG-LRU. Additionally, it motivates research into the potential benefits of using randomness for replacement policies. It is well-known that clever applications of randomness can provide cheap and consistent outcomes for various algorithms [25] Further, many page migration policies use random sampling approaches [7], [26]. Given the complex interactions between page table scans and replacement decisions, as well as their projected use in large datacenters, the use of principled randomness could provide an excellent tradeoff between the effectiveness and scalability of scanning heuristics.

D. Future Characterization

While this work characterizes MG-LRU under a large space of configurations, many important scenarios remain untested. In particular, all of our workloads have a memory footprint between 12-16GB. We have not tested more memory-intensive workloads, in which scanning efficiently becomes an even larger challenge as more pages reside in physical memory. Additionally, all of our experiments run a single application at a time, with a fresh reboot before each execution. While this helps isolate variance that results from the replacement policy, it does not test more realistic scenarios involving multiple running applications and fragmentation from long-running systems. Finally, we have not examined paging behavior under multi-tenancy, which induces additional complications through the presence of multiple containers or virtual machines. Intuitively, such settings would only increase variance due to factors such as interactions between host and guest operating systems, inconsistency in physical memory contiguity, application interleaving, and more. However, since paging policies should be performant and robust under all conditions, such additional characterizations are necessary as the field of memory management progresses.

VII. Conclusion

We conduct extensive experiments on MG-LRU spanning multiple workloads, parameter configurations, memory capacities, and swap mediums. Our experiments show that MG-LRU exhibits high variance across multiple axes. First, for a given workload and system configuration, MG-LRU exhibits high variation in total runtime and fault rates across otherwise identical workload executions. Second, across different system configurations, MG-LRU's performance is inconsistent under other parameter configurations and replacement policies. We find that a different policy is best for each workload in each tested system setup. Our results indicate that the quality of replacement decisions is a complex function of the method used to scan accessed bits, scanning overhead, promotion/migration policy, cost of swapping, memory footprint and working set size, memory capacity, and workload behavior. As applications become increasingly memory-hungry and memory hierarchies become increasingly tiered, significant work will be necessary to understand the nuances of page replacement and guide the evolution of future memory management policies.

VIII. Acknowledgements

We thank the anonymous reviewers for their comments and feedback. We also thank Nicholas Lindsay for our initial insight into MG-LRU. We also acknowledge the support of the National Science Foundation grant number 2112562 and a Graduate Research Fellowship.

References

[1] [Online]. Available: https://www.tpc.org/tpch/
[2] "Chapter 10 Page Frame Reclamation." [Online]. Available: https://www.kernel.org/doc/gorman/html/understand/understand013.html
[3] "Linux Statistics." [Online]. Available: https://99firms.com/blog/linux-statistics/#gref
[4] "Multi-Gen LRU." [Online]. Available: https://www.kernel.org/doc/html/v6.2/mm/multigen_lru.html
[5] "zram: Compressed RAM-based block devices." [Online]. Available: https://docs.kernel.org/admin-guide/blockdev/zram.html
[6] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking bandwidth for GPUs in CC-NUMA systems," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 354–365.
[7] N. Agarwal and T. 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*, 2017, pp. 631–644.
[8] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, 2015, pp. 1383–1394.
[9] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[10] R. W. Carr, "Virtual memory management." 1982.
[11] C.-H. Chang, J. Han, A. Sivasubramaniam, V. Sharma Mailthody, Z. Qureshi, and W.-M. Hwu, "GMT: GPU Orchestrated Memory Tiering for the Big Data Era," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 464–478.
[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.
[13] J. Corbet, "AutoNUMA: the other approach to NUMA scheduling," 2012. [Online]. Available: https://lwn.net/Articles/488709/
[14] ——, "Multi-generational LRU: the next generation," 2021. [Online]. Available: https://lwn.net/Articles/856931/
[15] ——, "The multi-generational LRU," 2021. [Online]. Available: https://lwn.net/Articles/851184/
[16] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[17] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 371–384.
[18] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.
[19] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 649–667.
[20] J. Hong, S. Cho, G. Park, W. Yang, Y.-H. Gong, and G. Kim, "Bandwidth-Effective DRAM Cache for GPU s with Storage-Class Memory," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 139–155.
[21] J. Kim, W. Choe, and J. Ahn, "Exploring the design space of page management for {Multi-Tiered} memory systems," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 715–728.
[22] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, "Mind: In-network memory management for disaggregated data centers," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 488–504.
[23] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. 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*, 2023, pp. 742–755.
[24] D. McCracken, "Object-based reverse mapping," in *Linux Symposium*, 2004, p. 357.
[25] R. Motwani and P. Raghavan, "Randomized algorithms," *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, pp. 33–37, 1996.
[26] J. Ren, D. Xu, J. Ryu, K. Shin, D. Kim, and D. Li, "MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 803–817.
[27] C. A. Waldspurger, "Memory resource management in vmware esx server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.
[28] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu, "Semeru: A {Memory-Disaggregated} managed runtime," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 261–280.
[29] J. Yang, Y. Yue, and R. Vinayak, "Segcache: a memory-efficient and scalable in-memory key-value cache for small objects," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 503–518.
[30] J. Yang, Y. Zhang, Z. Qiu, Y. Yue, and R. Vinayak, "Fifo queues are all you need for cache eviction," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 130–149.
[31] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *Osdi*, vol. 8, no. 4, 2008, p. 7.