PagePilot: Synergizing Heterogeneous Backend Devices with Reusability-Aware Page Offloading

XINGZE LIU, College of Computer Science, Nankai University, Tianjin, China
JIALIN DONG, Colledge of Conputer Science, Nankai University, Tianjin, China
XINYU LIU, College of Computer Science, Nankai University, Tianjin, China
LIZHI WANG, College of Computer Science, Nankai University, Tianjin, China
SHIYANG LI, Department of Computer Science and Engineering, University of Minnesota Twin Cities, Minneapolis, United States
HAORAN LI, College of Computer Science, Nankai University, Tianjin, China
XIAOLI GONG*, College of Computer Science, Nankai University, Tianjin, China
JIN ZHANG, College of Computer Science, Nankai University, Tianjin, China
PEN-CHUNG YEW, Department of Computer Science and Engineering, University of Minnesota Twin Cities, Minneapolis, United States

Page swapping remains the dominant mechanism for extending physical memory in modern operating systems. Despite the emergence of new memory technologies, swap-based memory extension continues to be widely deployed due to its transparency and compatibility with diverse storage backends. However, current Linux swap management relies on a simple priority-based policy when multiple heterogeneous backends are available, and is oblivious to page reuse behavior. As a result, fast devices are often occupied by pages that are rarely reused, while frequently refaulted pages are placed on slow devices, leading to unnecessary performance degradation under memory pressure. We present ***PagePilot***, a reuse-aware page offloading framework for heterogeneous swap systems. ***PagePilot*** persistently tracks refault behavior across eviction and refault cycles and uses average refault distance to guide backend selection. In addition, a background migration mechanism corrects misplacements to preserve fast-device capacity for pages with imminent reuse. We implement ***PagePilot*** in the Linux 6.3 kernel and evaluate it using representative data-serving and analytics workloads. Our results show that ***PagePilot*** improves application throughput and reduces page-fault handling latency under memory pressure, while substantially increasing the fraction of refaults served by fast backends.

CCS Concepts: • **Software and its engineering → Memory management**; **Virtual memory**; **Secondary storage**.

*Xiaoli Gong and Jin Zhang are the corresponding authors.

Authors' Contact Information: Xingze Liu, College of Computer Science, Nankai University, Tianjin, China; e-mail: liuxingze@mail.nankai.edu.cn; Jialin Dong, Colledge of Conputer Science, Nankai University, Tianjin, China; e-mail: jialindong200006@gmail.com; Xinyu Liu, College of Computer Science, Nankai University, Tianjin, China; e-mail: 2120250636@mail.nankai.edu.cn; Lizhi Wang, College of Computer Science, Nankai University, Tianjin, China; e-mail: nku_liz@icloud.com; Shiyang Li, Department of Computer Science and Engineering, University of Minnesota Twin Cities, Minneapolis, Minnesota, United States; e-mail: li004074@umn.edu; Haoran Li, College of Computer Science, Nankai University, Tianjin, China; e-mail: wanch_lhr@nankai.edu.cn; Xiaoli Gong, College of Computer Science, Nankai University, Tianjin, China; e-mail: gongxiaoli@nankai.edu.cn; Jin Zhang, College of Computer Science, Nankai University, Tianjin, China; e-mail: nkzhangjin@nankai.edu.cn; Pen-Chung Yew, Department of Computer Science and Engineering, University of Minnesota Twin Cities, Minneapolis, Minnesota, United States; e-mail: yew@umn.edu.

ACM 1544-3973/2026/9-ART
https://doi.org/10.1145/3845616

Additional Key Words and Phrases: Page Swapping, Virtual Memory, Heterogeneous Memory Systems, Reuse-Aware Placement, Page Fault Performance

1 Introduction

Memory is a critical resource that often falls short of meeting the demands of modern applications. For example, applications running on devices such as smartphones and laptops frequently encounter memory shortages due to constraints like cost, weight, and power consumption that limit the memory [23, 28] . Even on the server side, particularly in cloud environments that are often perceived as having virtually unlimited memory space, the challenge persists. The rising popularity of in-memory workloads such as machine learning applications and key-value stores is causing memory demands to grow even further [4]. Despite the advancements in hardware technologies, the imbalance persists between the growth in memory capacity and the increase in the number of CPU cores. This disparity is particularly pronounced in cloud centers where resources are provisioned as virtual machines by allocating CPU cores and memory together. This allocation model exacerbates memory shortages, especially on workloads with large memory footprints [12, 22, 33, 35].

Modern operating systems address memory capacity limitations by extending primary memory with secondary storage through *Page Swapping*. When physical memory is insufficient, pages are *evicted* from primary memory and *offloaded* to a backend device selected from available secondary storage resources. Despite decades of evolution in memory technologies, page swapping remains the dominant and most widely deployed mechanism for memory extension in production systems. While emerging technologies such as Non-Volatile Memory (NVM) and Compute Express Link (CXL) offer promising alternatives, their adoption in real-world deployments is still constrained by software maturity, system integration complexity, and deployment cost. Recent studies show that translating these technologies into stable, general-purpose memory extensions requires substantial engineering effort and ecosystem support, limiting their near-term applicability at scale [11, 15, 19]. In contrast, page swapping provides a transparent and portable abstraction that is compatible with a wide range of storage backends. For example, Remote Direct Memory Access (RDMA) enables high-bandwidth, low-latency access to memory resources on remote machines, commonly referred to as far memory or disaggregated memory. Existing systems leverage page swapping to integrate such remote memory into the virtual memory subsystem, exposing it through a familiar interface while achieving acceptable performance [1, 7, 21, 27, 30]. Moreover, mature storage technologies, including HDDs, SSDs, and compressed memory devices such as ZRAM, can readily serve as page-swapping backends, reinforcing the practicality, flexibility, and longevity of page swapping as a foundation for memory extension [3].

Multiple devices with different performance and capacity characteristics can be used as swap backends in the same system. A common configuration combines a relatively small, low-latency backend with a larger but slower backend, with the goal of providing both fast refault service and sufficient offloading capacity. The fast backend is typically capacity constrained because it may consume scarce DRAM resources or use storage that is impractical to provision at the same scale as the slower tier. Consequently, under sustained memory pressure, the offloaded footprint can exceed the capacity of the fast backend, forcing part of the swapped data onto the slower backend.

Linux zswap exemplifies this organization: it maintains a bounded compressed-memory pool and writes pages back to a backing swap device when that pool reaches its capacity [29]. Similarly, mobile systems combine ZRAM with flash storage to construct a two-level swap hierarchy [8, 14], while production memory offloading systems support backends ranging from compressed memory to NVMe SSDs [33]. Such configurations seek to obtain the latency benefit of the fast backend without being limited by its capacity.

However, backend device management in modern operating systems remains underdeveloped. In Linux, backend device management is primarily based on a priority list determined by the device bandwidth. This approach prioritizes the fastest device as the primary target for page offloading until its capacity is exhausted, and only then falls back to the next tier. While simple and straightforward, this strategy can result in unintended

behaviors. For instance, cold pages that are not accessed again may be offloaded to faster devices, leading to what is so-called *Fast Device Kidnapping*. Conversely, frequently accessed pages that are offloaded shortly after being swapped in may end up on slower devices, forcing a large number of refaults to be served by the slow swap devices, which is a phenomenon referred to as *Slow Device Refault Amplification*. We discuss both phenomena in Section 2.2. Our experiments showed that around 60% of swap slots on the fast devices are wasted due to these undesirable phenomena.

A natural way to mitigate this priority inversion is reuse-aware page offloading, which makes backend selection aware of whether an evicted page is likely to be reused soon. When a page that has been offloaded to a backend device is accessed again, a page fault occurs and triggers the operating system to swap the page back into main memory. In Linux, this event is referred to as a *refault*. Because the swap-in operation lies on the critical path of a refault, it is essential that pages likely to refault are placed on fast device whenever possible. Consequently, the choice of the target device for page offloading plays a crucial role in overall system performance. A well-established principle in tiered memory systems is that data with high temporal locality—i.e., a short reuse distance—should be preferentially placed on faster storage device to minimize access latency. In the context of page swapping, data reuse can be naturally observed through refault events, and the corresponding refault distance serves as an effective indicator of reuse behavior. Leveraging refault distance history information to guide backend device selection during page offloading enables reuse-aware placement decisions that reduce refault latency and improve system efficiency.

However, tracing the refault history of an individual page is impractical in the current Linux swap system. During page swapping, a victim page is unmapped from the process's virtual address space and offloaded to a backend device using an identifier assigned by the swap entry management subsystem (see Section 2.1). While this design keeps the swap interface simple and subsystems loosely coupled, it discards virtual memory semantics and access history at eviction. Furthermore, metadata associated with swapped-out pages is lost after refault, and swap entry allocation does not guarantee stable identifiers across swap cycles. As a result, accurately tracking page reuse and predicting reusability are impractical.

In this paper, we propose ***PagePilot***, a framework that makes reuse-aware placement decisions for heterogeneous swap backends during page offloading. ***PagePilot*** incorporates an *Offloading Tracer* that bridges the semantic gap between the virtual memory subsystem and swap backends by capturing page swapping history. An *Offloading Router* then applies a heuristic prediction algorithm to estimate page reusability and selects appropriate backend devices based on the expected in-swap duration of each page. To mitigate the impact of prediction errors, ***PagePilot*** further includes a *Backend Migrator* that dynamically adjusts page placement across backend devices, ensuring that the fast device retains sufficient free capacity.

We implemented a prototype of ***PagePilot*** in the Linux kernel (v6.3) and conducted comprehensive experiments using application workloads derived from real-world scenarios. The implementation is publicly available at https://github.com/NKU-EmbeddedSystem/pagepilot. The results show that ***PagePilot*** improves performance under memory pressure for data-intensive applications, achieving an average reduction in page-fault handler latency of 10.8% and a 51.8% relative increase in fast device utilization. ***PagePilot*** also outperforms iSwap, a recent swap optimization, in 37 of the 40 configurations, with a geometric-mean throughput advantage of 23.6%.

In summary, we have made the following contributions:

- We identify a fundamental limitation in existing Linux swap systems: the lack of reuse-aware placement across heterogeneous swap backends, which leads to unnecessary swap-in latency and underutilization of fast storage devices under memory pressure.
- We propose ***PagePilot***, a reuse-aware swap management framework that bridges the semantic gap between the virtual memory subsystem and heterogeneous swap backends. Its *Offloading Router* guides backend

placement using page reuse history, while background migration corrects mispredictions and preserves fast-device capacity.
- We implement ***PagePilot*** as a prototype in the Linux kernel and evaluate it using data-intensive workloads derived from real-world scenarios. Experimental results show that ***PagePilot*** achieves an average reduction in operation latency of 10.8% and a 51.8% relative increase in fast device utilization under memory pressure.

The rest of the paper is organized as follows. Section 2 describes the background and motivation by explaining the shortcomings of existing swap systems. Section 3 presents the design and implementation of ***PagePilot***. Section 4 evaluates the performance of ***PagePilot***. Section 6 discusses related work and Section 7 concludes the paper.

2 Background and Motivation

2.1 Page Swap in Modern Systems

Page swapping is a fundamental mechanism in the virtual memory subsystem of modern operating systems. When physical memory becomes insufficient, the kernel reclaims space by evicting selected pages to a secondary storage device and later retrieves them on demand when they are accessed again. This mechanism allows the system to preserve the virtual-memory abstraction while operating beyond the limits of installed DRAM [6]. Page swapping is widely relied upon in resource-constrained environments such as mobile and embedded systems, where memory capacity is limited relative to workload demands [3, 39]. Swap activity is also common in systems that appear to have sufficient physical memory. In these environments, eviction and refault behavior are often exercised intentionally as part of memory diagnosis and resource management. For example, the widely deployed ballooning mechanism in virtualized cloud infrastructures induces controlled memory pressure and monitors swap and refault activity to infer a VM's effective memory demand [24]. Similar techniques have been reported in production systems such as TMO, where swap behavior is used to characterize memory pressure and guide workload placement decisions [33].

The page swapping process can be conceptually divided into two stages. Figure 1 illustrates the procedure of swap out. The first stage, *Page Eviction*, occurs when the system is under memory pressure and the kernel selects pages to be reclaimed from physical memory. The second stage, *Page Offloading*, transfers these eviction victims to a secondary storage device, which we refer to as a *backend device*. If an evicted page is accessed again, it is then swapped in. A page fault is triggered and the page is brought back into memory on demand; this event is referred to as a *Refault*.

Because a refault lies on the critical path of memory access, it is essential to avoid evicting pages that are likely to be referenced again. A substantial body of prior work has therefore focused on improving eviction accuracy by predicting which pages are least likely to be reused during the *Page Eviction* stage. Representative examples include MGLRU [9], iSwap [32], Hemem [25], Nimble [36], and Memtis [13]. Most of these approaches rely on historical access behavior to estimate future reuse. As illustrated in Figure 1, access frequency (often captured as a "temperature" metric) is used as an indicator for eviction priority: "cold" pages, i.e., those with long reuse distances, are preferentially selected as eviction candidates and offloaded to the backend devices.

The *page offloading* stage has received comparatively little attention, as the swap-slot allocation mechanism is simple and straightforward. When a free location is available on a backend device, the kernel allocates a swap slot and transfers the contents of the victim page to that slot. Even for the situation when multiple backend devices with different bandwidth and latency characteristics are present, the Linux kernel adopts a priority-based strategy for device selection [2]. More specifically, as illustrated in Figure 1, offloading backend devices are organized as a priority list, with priorities assigned based on the performance of the devices. Higher-performing devices are assigned a higher priority. The swap slot is then allocated from the highest-priority device that has available space. By analogy with eviction policies, pages with higher reuse likelihood should ideally reside on

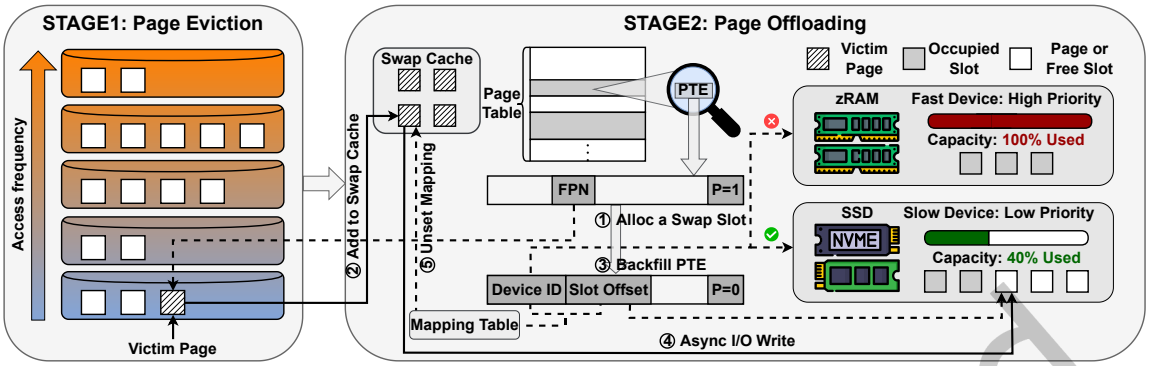


Fig. 1. **The workflow of page eviction and offloading in the Linux kernel.** The process consists of selecting a victim page based on access frequency (Stage 1) and offloading it to swap storage (Stage 2). The kernel allocates a swap slot from the highest-priority available device (①), buffers the page in the swap cache (②), and updates the Page Table Entry (PTE) with a swap identifier, which establishes a mapping to the swap cache via a mapping table to handle concurrent access (③). Subsequently, the page is asynchronously written to the backend device (④), and the mapping to swap cache is replaced by a shadow entry and the page in the swap cache is reclaimed (⑤).

faster devices. Unfortunately, priority-based slot allocation is oblivious to reuse behavior and therefore cannot consistently preserve fast device capacity for pages that are likely to refault.

However, tracing page access history and predicting reuse in the swap system is impractical due to the decoupled design of the Linux kernel shown in Figure 1. Once a victim page is selected during the eviction stage, the kernel first allocates a swap-entry identifier for the page (Step ①) and creates a swap entry that specifies the backend device location associated with that identifier. In Step ②, the victim page is moved into the swap cache. And then, in Step ③, the victim page is unmapped from the virtual address space by marking its PTE as non-present and the swap-entry identifier is written into the PTE. And at this point, the `Device ID` and `Slot Offset` in the PTE can be used to establish a mapping to the swap cache via a mapping table, ensuring concurrent refault while page writeback is in flight. Notably, although the page data still resides in DRAM, its access history and virtual-memory semantics have already been discarded. In Step ④, the page contents are asynchronously written to the backend device, and in Step ⑤ the page is finally reclaimed after the write completes and the mapping table item is updated to a non-page value (e.g., a shadow entry) that records backend residency without retaining an in-memory page pointer. During a refault, the identifier stored in the PTE is used to locate the page on the backend device and retrieve it back into memory. After the swap-in completes, the identifier and its associated metadata are discarded. Crucially, the swap-entry identifier has no persistent relationship to the virtual page: even if the same page is swapped out multiple times, there is no guarantee that it will receive the same identifier. As a result, it is impossible to reconstruct access history or reuse behavior by tracking swap-entry identifiers across offloading and refault events.

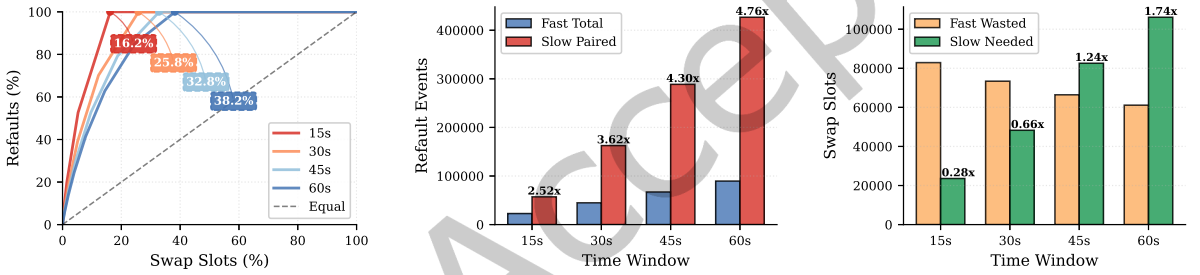
2.2 Priority Inversion in Page Offloading

Without reuse-aware information, a priority-based backend selection policy can lead to undesirable offloading behavior under memory pressure. When the number of offloaded pages is small, most pages can be placed on the highest-performing device, and the policy works reasonably well. However, once the number of offloaded pages exceeds the capacity of the fastest device, pages with long refault distances may occupy the high-performance

device, while pages with short expected refault distances are forced onto slower device. We refer to this pathological placement as *Priority Inversion in Page Offloading*.

To study the performance impact of priority inversion, we conduct a set of experiments using the Redis key–value store running a YCSB workload [5]. The workload follows a skewed access pattern in which 80% of requests target 20% of the keys. The experiments are performed on a server equipped with an Intel Xeon® Gold 6442Y CPU, running Linux kernel v6.3. In our setup, we configure a ZRAM as the fast device with capacity for 410 MB of uncompressed page data, while an SSD with sufficient capacity is used as the slow device. The Redis working set is configured to 4 GB, and the available physical memory is restricted to 410 MB to induce sustained memory pressure.

We record the page faults within a fixed time window and analyze the swap slots involved in those refaults. For the fast device, we construct a cumulative distribution function (CDF) over swap slots for each window, where the x-axis represents the fraction of swap slots on the fast device and the y-axis represents the fraction of page faults they account for. We evaluate time windows of 15 s, 30 s, 45 s, and 60 s; the results are shown in Figure 2a. Across all window sizes, the distribution is highly skewed. For example, 16.2% of swap slots account for nearly all refaults for the 15 s window test, while even over a 60 s window, approximately 61.8% of fast device slots experience no refaults at all. This indicates that a substantial portion of the fast device is occupied by pages that provide no refault-serving benefit during the observation period.



(a) Fast device kidnapping.

(b) Refault comparison: Fast vs. Slow.

(c) Wasted fast slots vs. needed slow slots.

Fig. 2. **Fast device kidnapping and slow device refault amplification.** Experiments were conducted using Redis with a skewed YCSB workload. (a) The CDF of fast device usage reveals that a small minority of slots (e.g., 16.2% in the 15s window) account for nearly all refaults, indicating that the majority of high-speed capacity is "kidnapped" by cold pages. (b) Consequently, the slow device handles significantly more refault events (2.52×–4.76×) than the fast device, demonstrating severe priority inversion. (c) A comparison between the wasted fast device slots (*Fast Wasted*) and the slot demand of hot pages on the slow device (*Slow Needed*) shows that sufficient fast device capacity exists to accommodate the frequently refaulted pages, highlighting a clear optimization opportunity.

We also analyze paired *eviction–refault* events on the slow device, where a pair denotes a page that is evicted and later refaults back into memory. As shown in Figure 2b, the number of paired eviction–refault events served by the slow device (red bars) is significantly higher than all refault events served by the fast device (blue bars) in the same time window, by a factor of 2.52×–4.76×.

For the pairs served by the slow device, we further perform a timestamp-based interval analysis. A swap slot is considered occupied for the duration between the eviction time and the subsequent refault time. Two eviction–refault pairs can share the same swap slot if their occupancy intervals do not overlap; otherwise, an additional swap slot is required. We compute the maximum number of swap slots needed to serve these pairs

with time overlap and report the values as the green bars in Figure 2c. For comparison, we also report the number of swap slots on the fast device that never serve any refault during the same period (orange bars), representing wasted fast device capacity. We observe that, in most cases, the slow device eviction–refault activity could have been accommodated by these wasted fast device slots. This result reveals a clear optimization opportunity: fast-device capacity assigned to pages that do not refault during the observation window could instead be assigned to pages that subsequently refault from the slow device.

The preceding experiment uses a 10% memory ratio to clearly expose the placement inefficiency. To examine how this inefficiency changes as memory pressure decreases, we repeat the same analysis while increasing the available DRAM to 30%, 50%, 70%, and 90% of the Redis working set. The fast-device capacity remains fixed across all configurations.

In addition to *Fast Wasted* and *Slow Needed*, we define the *placement opportunity* as $\min(\textit{Fast Wasted}, \textit{Slow Needed})/C_{\text{fast}}$, where C_{fast} is the fast-device capacity. This metric reports the fraction of the fast device involved in a potentially beneficial reassignment. For example, if *Fast Wasted* is larger than *Slow Needed*, all observed slow-device demand can be accommodated by the ineffective fast-device slots. The placement opportunity is nevertheless $\textit{Slow Needed}/C_{\text{fast}}$, rather than 100%, because it measures the fraction of the fast-device capacity that can be assigned more effectively, rather than the fraction of slow-device demand that can be accommodated.

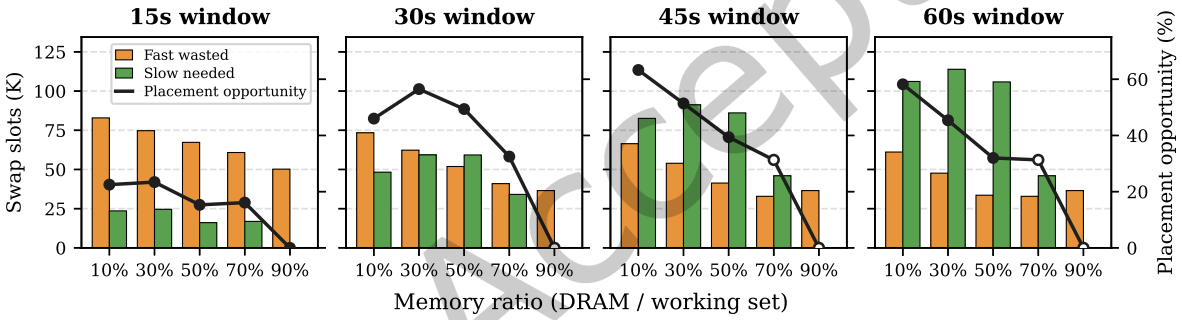


Fig. 3. **Fast-device waste and placement opportunity across memory ratios.** Bars show occupied fast-device slots that serve no refault (*Fast Wasted*) and the maximum concurrent slot demand of slow-device eviction–refault pairs after accounting for temporal slot reuse (*Slow Needed*). The line shows the placement opportunity, defined as $\min(\textit{Fast Wasted}, \textit{Slow Needed})$ normalized by the fast-device capacity. Open markers identify groups whose post-snapshot execution is shorter than the nominal observation window; both bars and the line for these groups use the full remaining trace.

Figure 3 shows that the available placement opportunity generally decreases as more DRAM becomes available. In the 60 s window, it decreases from 58.2% at the 10% memory ratio to 45.4% and 32.0% at the 30% and 50% ratios, respectively. At the 70% ratio, the full remaining post-snapshot trace still exposes a 31.3% opportunity, while the opportunity reaches zero at the 90% ratio. As the memory ratio increases, fewer pages need to remain outside DRAM, allowing the fixed-capacity fast device to cover a larger fraction of the relevant offloaded pages. Consequently, fewer pages that subsequently refault depend on the slow device, and the benefit available from reuse-aware backend placement diminishes.

The observation window captures a complementary temporal effect. A page that does not refault within a short window may still refault later. As the window increases, more fast-device slots become active and are no longer classified as wasted. For example, at the 10% memory ratio, the active fraction of the fast device increases from 16.2% in the 15 s window to 25.8%, 32.8%, and 38.2% in the 30 s, 45 s, and 60 s windows, respectively. Meanwhile,

the longer window exposes additional slow-device eviction–refault pairs and increases *Slow Needed*. The four panels therefore characterize the same placement inefficiency over progressively longer reuse horizons.

The placement inefficiency is not specific to skewed key–value accesses. We also examine an HNSW-based approximate nearest-neighbor query workload over the BIGANN dataset [10, 20]. Using `hnswlib`, we build an in-memory HNSW index over 10 million 128-dimensional vectors. The steady-state footprint of the index and query pool is approximately 8.4 GiB.

Sixteen query threads issue 200,000 distinct queries sampled uniformly without replacement from the two-million-vector BIGANN learn set. HNSW searches traverse shared upper graph layers before descending toward query-specific neighborhoods, causing independent queries to revisit common portions of the index. This graph-induced reuse differs from the key-popularity-driven access pattern in Redis.

We configure a 1 GiB ZRAM fast backend and a 16 GiB NVMe slow backend, preserving approximately the same 1:10 fast-capacity-to-footprint ratio used in the Redis experiment. All index, search, and query parameters remain fixed across memory ratios. For HNSW, the memory ratio is normalized to the 11.3 GiB peak footprint observed during index loading, reflecting the memory capacity that must be provisioned to avoid load-time thrashing.

Table 1 summarizes the placement opportunity for both workloads. HNSW exhibits the same overall trend as Redis: its placement opportunity decreases from 34.0% at the 10% memory ratio to 31.0% and 26.5% at the 30% and 50% ratios, respectively. Thus, substantial placement opportunity remains under moderate memory pressure even though HNSW derives reuse from a fundamentally different access pattern.

Table 1. **Placement opportunity across workloads and memory ratios.** Each value reports the fraction of fast-backend capacity involved in a potentially beneficial reassignment over the 60 s observation window.

	Memory ratio				
Workload	10%	30%	50%	70%	90%
Redis	58.2%	45.4%	32.0%	31.3%[†]	0
HNSW	34.0%	31.0%	26.5%	0	0

[†]The Redis execution ends before the nominal 60 s window; the value uses the full remaining post-snapshot trace.

As the available DRAM increases, the steady-state evicted set shrinks relative to the fixed-capacity fast backend, reducing the need to place subsequently reused pages on the slow device. For HNSW, the evicted set at the 70% ratio is 788 MiB and therefore fits within the 1.1 GiB fast backend; at the 90% ratio, the workload generates no swap traffic. Redis reaches the same capacity boundary at the 90% ratio. Although the nominal transition point differs because HNSW's load-phase peak includes transient index-loading state, both workloads exhibit the same underlying behavior: the placement opportunity disappears once the steady-state evicted set no longer exceeds the fast-backend capacity.

Together, the Redis and HNSW results show that priority inversion is a general consequence of reuse-oblivious backend selection rather than an artifact of a particular application or access distribution. Under memory pressure, offloaded pages exhibit heterogeneous reuse behavior, and the limited fast-backend capacity must be assigned according to their expected refault behavior rather than allocation order.

Motivated by these observations, we propose ***PagePilot***, which preserves page reuse history across eviction and refault, uses this history to guide backend selection during page offloading, and corrects persistent misplacements through backend migration.

3 *PagePilot* Design and Implementation

3.1 Overview of *PagePilot*

In this section, we propose ***PagePilot***, which bridges the runtime information gap between the virtual memory management module and the offloaded page management module, and employs heuristic methods to dynamically predict page placement during offloading. Additionally, ***PagePilot*** integrates a background detector and migrator to mitigate the impact of incorrect predictions, ensuring adaptive and efficient memory management.

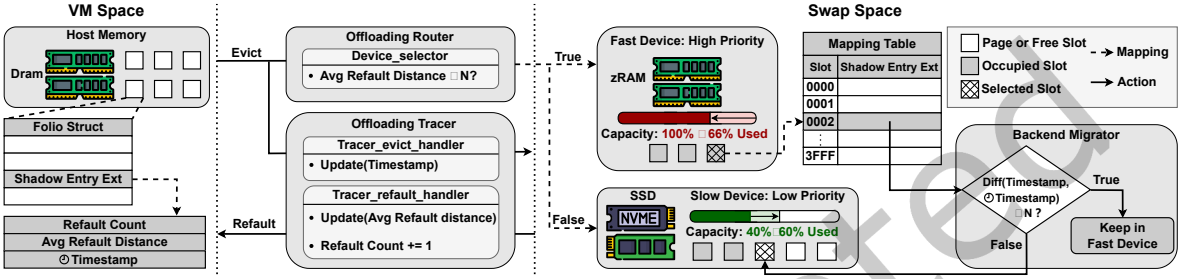


Fig. 4. **Overview of *PagePilot*.** The framework optimizes hybrid swap space through three components: (1) The **Offloading Tracer** extends the folio structure and shadow entries to record metadata for bridging the semantic gap between VM Space and Swap Space; (2) The **Offloading Router** directs evicted pages to the High-Priority Fast Device (zRAM) or Low-Priority Slow Device (SSD) based on the average refault distance; and (3) The **Backend Migrator** asynchronously identifies and relocates long-resident cold pages from the Fast Device to the Slow Device to reclaim capacity for hot pages.

As shown in Figure 4, the proposed framework consists of three key components. The *Offloading Tracer* bridges the semantic gap between the virtual memory subsystem and the swap backends by maintaining the correspondence between virtual-memory pages (represented as `folios` in the Linux kernel) and their associated swap entries. For each evicted page, the tracer records the offloading timestamp, refault count, and average refault distance, and stores a reference to this metadata in the corresponding folio structure. The *Offloading Router* uses this information to predict appropriate placement decisions for evicted pages. In particular, the average refault distance serves as the primary feature for backend selection, while the router dynamically adjusts its criteria based on current memory pressure and backend device utilization. Finally, the *Backend Migrator* monitors page behavior and relocates misclassified pages. Specifically, it identifies pages that remain on the fast device for extended periods without refaulting and migrates them to slower devices, thereby reclaiming fast device capacity for frequently refaulted pages.

For clarity of exposition, we focus on systems with two types of swap backends. In practice, two devices are sufficient in many deployments where swap serves as a supplementary extension to physical memory—for example, configurations such as ZRAM+SSD or SSD+HDD. In contrast, systems that use swap as a true memory expander may involve more than two backends, such as disaggregated memory pools with heterogeneous latency and bandwidth characteristics. Our current implementation of ***PagePilot*** supports two backend types, but the design generalizes naturally and can be extended to additional tiers with minimal modification.

3.2 Offloading Tracer

The *Offloading Tracer* maintains a persistent metadata object, referred to as a *Shadow Entry Ext*, for each page across successive eviction and refault cycles. When the page resides in DRAM, the structure is reachable through its in-memory folio; after offloading, it is anchored in the swap space via a per-device `Mapping Table`. In this

manner, refault history remains stable across multiple iterations of eviction and refault. The tracer records three fields: the eviction `Timestamp`, the `Refault Count`, and the `Avg Refault Distance`. `Timestamp` is updated at each eviction, while `Avg Refault Distance` and `Refault Count` are updated upon refaults.

During the first eviction of a page, `tracer_evict_handler` allocates a new *Shadow Entry Ext*, initializes `Timestamp` with the current timestamp, and attaches the structure to the corresponding per-device `Mapping Table`. On a refault, the kernel resolves the swap entry, reaches the associated *Shadow Entry Ext*, and `tracer_refault_handler` increments `Refault Count` and updates `Avg Refault Distance` based on the elapsed time since `Timestamp`; the structure is then reattached to the folio created for the refaulted page.

Notably, the number of offloaded pages during execution can far exceed the working set size, and retaining metadata for all pages would incur unnecessary memory overhead. To bound tracking cost, the *Offloading Tracer* selectively maintains history only for pages that exhibit frequent eviction–refault activity. Specifically, a *Shadow Entry Ext* is discarded when a page remains in the backend device beyond a predefined timeout interval, indicating that it is unlikely to refault again. This adaptive retention policy ensures that only pages relevant to reuse prediction are tracked, while keeping memory overhead modest.

3.3 Offloading Router

The *Offloading Router* translates the reuse history maintained by the *Offloading Tracer* into an initial backend-placement decision for each evicted page. It is invoked during page offloading, after the existing reclamation policy has selected a victim page and before the swap backend is chosen. The router retrieves the page's historical average refault distance from its *Shadow Entry Ext*. Therefore, the router changes only the offloading destination and remains independent of victim-page selection.

Rather than relying on a heavyweight predictor, ***PagePilot*** uses a page's historical average refault distance as a compact indicator of its reuse likelihood. A smaller refault distance indicates a higher likelihood of near-term reuse. Let $\overline{D}_p$ denote the historical average refault distance of page p, and let N denote the configurable admission threshold. The routing decision is expressed as follows:

$$\text{route}(p) = \begin{cases} \textit{Fast}, & \text{valid}(p) \wedge \overline{D}_p \le N, \\ \textit{Slow}, & \text{otherwise}, \end{cases} \tag{1}$$

where $\text{valid}(p)$ indicates that the page has completed at least one eviction–refault cycle and therefore has a valid historical average refault distance. During the first eviction of a page, the *Offloading Tracer* has only created its *Shadow Entry Ext* and initialized the eviction timestamp; no refault observation is available at this point. ***PagePilot*** conservatively routes such a page to the slow device. This policy assumes that only a small fraction of newly evicted pages will be immediately reused and, more importantly, prevents pages without any reuse evidence from unconditionally consuming the limited capacity of the fast device. The routing decision consequently requires only a metadata lookup, a history-validity check, and a threshold comparison.

The threshold N is selected according to workload behavior and the capacity of the fast device. ***PagePilot*** determines its value during a profiling phase performed once for each workload configuration. During profiling, fast-device occupancy is treated as a control signal, and N is adjusted to maintain occupancy within a target operating region. The adjustment process follows a feedback pattern similar to PID-style control [40]: after every fixed number of evictions, the profiler samples the current fast-device occupancy and determines which watermark interval it falls into. If the fast device is underutilized, the profiler increases N to admit more pages; if the device is overutilized, the profiler decreases N to reserve capacity for pages with stronger near-term reuse. The adjustment uses predefined occupancy watermarks and bounded incremental updates, with persistent deviations from the target region triggering progressively stronger correction. The accumulated adjustment state is reset when the occupancy moves to a different watermark interval. This process continues until occupancy

stabilizes within the desired high-utilization band, at which point profiling terminates and the resulting value of N is recorded.

The profiling phase serves as a cold-start procedure. In our evaluation, profiling is performed before the measured run, after which the learned N is fixed. Once learned, N can be reused across subsequent runs of the same workload configuration. ***PagePilot*** can also optionally re-enable threshold adjustment when workload characteristics change, enabling adaptive behavior under changing access patterns. Since the router determines only the initial placement, prediction errors may still occur. The *Backend Migrator* described in Section 3.4 subsequently corrects pages that remain on the fast device substantially longer than expected.

3.4 Backend Migrator

Because building a perfect predictor is impractical, misclassification by the *Offloading Router* may still lead to **Priority Inversion in Page Offloading**, as described in Section 2.2. To mitigate this issue, the *Backend Migrator* runs as a background service. Backend migration is activated according to fast-device occupancy rather than a fixed wall-clock period, and the corresponding occupancy thresholds are configurable. It identifies stale pages on the fast device that have not refaulted for an extended period, using the offloading and refault timestamps maintained by the *Offloading Tracer*. Such pages are then asynchronously migrated to a slower device, freeing fast device capacity and improving overall efficiency.

The *Backend Migrator* generally applies the same admission threshold used by the *Offloading Router*, but in reverse: it targets pages whose observed reuse distance has grown substantially longer than expected. In configurations where premature migration is more costly, this threshold can be conservatively relaxed so that a page is moved to the slow device only after stronger evidence of low reuse. During each scan interval, the migrator inspects fast device residents, consults the metadata stored in their *Shadow Entry Ext*, and relocates over-staying pages to the slow device to restore fast device headroom for frequently refaulted pages. Because swap backends typically do not support direct device-to-device transfer, migration is performed via DRAM. ***PagePilot*** first pulls the selected page into memory, then writes it to the slow device while allocating a new destination slot.

The primary challenge in supporting the *Backend Migrator* stems from the decoupled design of swap entry management in the Linux kernel, as discussed in Section 2.1. When a page is offloaded, a swap entry is assigned to indicate its storage location, and this swap entry is stored in the corresponding PTE. To migrate an offloaded page to a new location, a new swap entry must be generated. However, scanning all page tables to update the swap entry is impractical due to efficiency constraints.

To address this issue, we introduce an indirect swap entry table, as illustrated in Figure 5. This table maintains a mapping between old and new swap entries after migration. During a refault event, the swap entry stored in the PTE is used to query the table for the updated location. If the page has been migrated, the new swap entry is retrieved and used to locate the page. Otherwise, if the lookup fails, the original swap entry is used to retrieve the offloaded page. A radix-tree-based data structure (`xarray` in the Linux kernel) is employed to optimize lookup efficiency.

Once an offloaded page on the fast device is migrated to a slower backend, its previous storage position is marked as available in the bitmap used for backend device management. To efficiently reuse the space freed after migration, we extend the Linux kernel's swap entry management with version control. The version prevents a stale swap entry from aliasing a new page when a freed source slot is reused before the original page refaults.

Instead of using the `Slot Offset` solely as the location index on the swap backend, we repurpose a portion of its high-order bits to encode a *Version*, as illustrated in Figure 5. During swap entry allocation, once an available position (denoted as *offset* in the figure) is identified in the backend device, the `Indirect Swap Entry Table` is

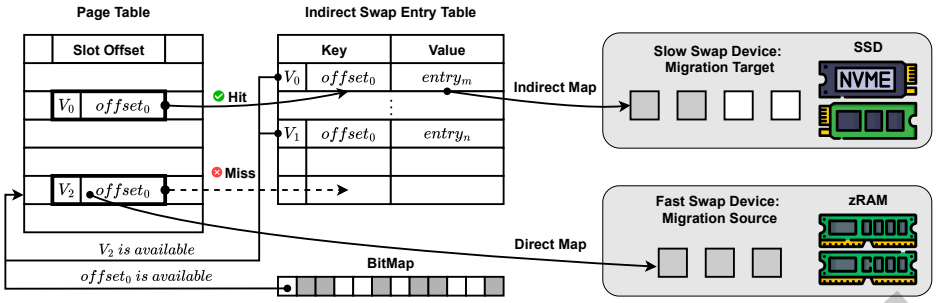


Fig. 5. **Indirect swap-entry remapping with versioned slot reuse.** To support backend migration without rewriting PTEs, we introduce an *Indirect Swap Entry Table* that maps an original swap entry to an updated entry after migration; refaults first consult this table and fall back to direct lookup when no remapping exists. Freed fast device offsets are tracked in the device bitmap and can be reused by encoding a small *Version* in the high-order bits of the slot offset.

scanned to obtain a version number that has not been previously used for this position. The offset and version number are then combined to generate the swap entry, which is subsequently stored in the corresponding PTE.

Given the limited number of bits available in the PTE for storing `Slot Offset`, we allocate only 2 bits for the version number, allowing a maximum of 4 reuse cycles for a specific location in the fast device. Since the version number is used to track continuous migrations at the same location, we consider a 2-bit allocation sufficient, as such events are infrequent. Our experimental results indicate that multiple reuses (i.e., reuse count > 1) were nonexistent in our test scenarios.

Even in the rare case where all version numbers for a particular location are exhausted, the impact remains minimal. In such a scenario, we simply skip the occupied swap position and select an alternative location in the fast device, which does not lead to significant performance degradation or system instability.

4 Evaluation

4.1 Experimental Setup

PagePilot is implemented on the `linux-xanmod-x64v1-v6.3` kernel running *Ubuntu Server 22.04.5*. The testbed is equipped with an Intel Xeon® Gold 6442Y CPU, 256 GiB of DDR5 memory, and a 4 TB M.2 NVMe SSD. ZRAM serves as the fast swap backend, while an NVMe swap partition serves as the slow backend. Across all workloads, the fast-backend capacity is approximately one tenth of the application memory footprint.

The workload suite contains two in-memory key–value stores and one approximate-nearest-neighbor search workload. Redis and Memcached are driven by YCSB [5], each with a 32 GiB working set and 24 million operations. Both Hotspot and Zipfian distributions are included; in each distribution, approximately 80% of requests access 20% of the keys. The HNSW workload uses `hnswlib` 0.8.0 and the BIGANN dataset [10, 20]. Its prebuilt index contains the first 10 million 128-dimensional vectors from the BIGANN base set. Sixteen threads issue 200,000 distinct queries selected uniformly from the official two-million-vector BIGANN learn set. The index uses the L2 distance metric with $M = 16$ and $ef_{\text{construction}} = 100$; searches use $ef_{\text{search}} = 64$ and $k = 10$. The same index and query sequence are used in every configuration. HNSW has an approximately 8.4 GiB steady-state memory footprint and an 11.3 GiB load-phase peak, with its memory ratios defined relative to the latter. The prebuilt index is loaded before the target DRAM limit is applied. The unmodified `linux-xanmod-x64v1-v6.3` kernel serves as the Linux baseline and follows the default priority-based multi-backend swap policy described in Section 2.2. We additionally compare against iSwap [32], which uses sampled page-access information to improve victim selection but does not alter swap-backend selection. Like the Linux baseline, iSwap therefore relies on the default

Table 2. Workloads and memory configurations. DRAM is varied from 10% to 80% of the listed footprint in increments of 10 percentage points. The HNSW footprint denotes its load-phase peak; the slow capacity denotes the per-workload NVMe-backed swap limit.

	Memory configuration			Workload configuration	
Workload	Footprint	DRAM range	Fast / slow	Access pattern	Requests
Redis	32 GiB	3.2–25.6 GiB	3.2 / 32 GiB	Hotspot, Zipfian	24 M each
Memcached	32 GiB	3.2–25.6 GiB	3.2 / 32 GiB	Hotspot, Zipfian	24 M each
HNSW	11.3 GiB	1129–9035 MiB	1.1 / 16 GiB	Uniform	200 K

priority-based policy to select the backend for each evicted page. All systems use identical workload inputs, DRAM limits, and swap-backend configurations.

Available DRAM ranges from 10% to 80% of each workload's memory footprint in increments of 10 percentage points. Table 2 summarizes the workload and memory configurations.

4.2 Performance Improvement and Upper-Bound Analysis

In this section, *Baseline* denotes the unmodified `linux-xanmod-x64v1-v6.3` kernel and its default priority-based multi-backend swap policy. By improving the selection of offloading destinations, ***PagePilot*** serves more demand refaults from the fast backend, thereby reducing refault latency and improving application performance under memory pressure. We first quantify this benefit using throughput normalized to Baseline. Because each experiment executes a fixed number of YCSB operations or HNSW queries, end-to-end completion time is inversely proportional to throughput; the corresponding absolute throughput and completion time are reported in the subsequent performance breakdown.

After presenting the end-to-end comparison, we examine why ***PagePilot*** improves performance over Baseline. Specifically, we measure the fraction of refaults served by the fast backend and use `ftrace` [26] to analyze average refault latency. We also derive a placement upper bound from the complete eviction–refault trace under the same fast-backend capacity, allowing us to compare online placement with an ideal global assignment. We further perform Coverage-p replays to assess how strongly the upper bound depends on complete foresight.

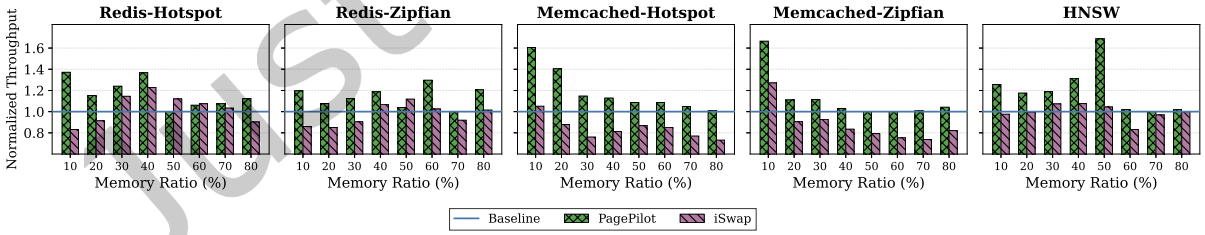


Fig. 6. **Normalized throughput.** Throughput of ***PagePilot*** and iSwap normalized to Baseline for Redis, Memcached, and HNSW as the memory ratio varies from 10% to 80% (higher is better). Baseline is normalized to 1.0. All subplots share the same y-axis scale.

End-to-end performance. Figure 6 summarizes the application-level performance of Baseline, ***PagePilot***, and iSwap. Across all 40 workload and memory configurations, ***PagePilot*** improves throughput over Baseline by 15.5% on geometric average and by up to 68.7%. The largest gain occurs for HNSW at the 50% memory ratio, where

throughput increases from 883 to 1,490 queries/s. ***PagePilot*** also outperforms iSwap in 37 of the 40 configurations, with a geometric-mean throughput advantage of 23.6%.

Because each run processes a fixed amount of work, these throughput improvements correspond to a 13.4% geometric-mean reduction in end-to-end completion time and a maximum reduction of 40.7%. The benefits are generally largest under severe to moderate memory pressure, where frequent refaults make backend placement directly affect application-visible fault stalls.

As the memory ratio increases, frequently and repeatedly accessed pages are more likely to remain resident in DRAM rather than being selected by reclaim and swapped out. The remaining swapped pages therefore tend to exhibit weaker or less stable future reuse, making their placement value more difficult for an online policy to infer from past behavior alone. Consequently, the end-to-end benefit of ***PagePilot*** gradually decreases. This trend does not imply that backend placement is no longer important: as shown below, the global Oracle constructed from the complete trace still reveals substantial gaps between ideal and online placement in several higher-memory configurations.

Capacity-constrained placement upper bound. We construct a global placement Oracle from the complete eviction–refault trace. Each pair defines an interval

$$I_i = [t_i^{\text{evict}}, t_i^{\text{refault}}),$$

during which placing the page on the fast backend occupies one slot. Given C slots, the Oracle scans intervals in eviction-time order. Whenever more than C selected intervals overlap, it removes the one with the latest refault time. For equal-sized 4 KiB pages, this rule maximizes the number of refaults served by the fast backend and therefore gives the exact trace-constrained placement upper bound.

Let F_{oracle} denote the resulting fast-backend refault ratio. Its corresponding ideal latency is estimated as

$$\widehat{L}_{\text{oracle}} = F_{\text{oracle}} L_{\text{fast}} + (1 - F_{\text{oracle}}) L_{\text{slow}},$$

where L_{fast} and L_{slow} are the backend-specific mean refault latencies measured for Baseline.

Coverage-p replay. To quantify the Oracle's dependence on future information, each interval is independently marked as informed with probability p. An informed interval exposes its exact refault time; an uninformed interval is admitted only when a slot is available and cannot subsequently be displaced. When the fast backend is full, an informed arrival replaces the informed resident with the latest refault time only if it refaults earlier. Thus, $p = 1$ recovers the full Oracle. We report the results for $p = 0.5$ and $p = 0.8$. These no-migration replays are sensitivity points rather than additional optimal upper bounds. Their latency estimates replace F_{oracle} in the equation above with the replayed fast-backend refault ratio F_p.

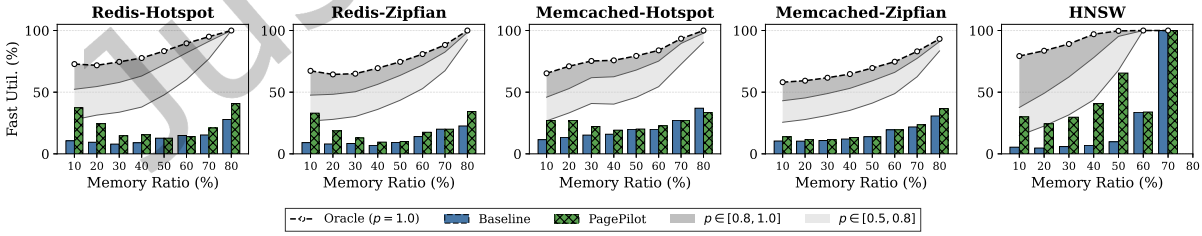


Fig. 7. **Fast-backend refault ratio.** Bars show Baseline and ***PagePilot***; the dashed line is the full-information Oracle ($p = 1.0$). The dark and light regions span the Coverage-p replay results for $p \in [0.8, 1.0]$ and $p \in [0.5, 0.8]$, respectively. HNSW at 80% incurs no refaults.

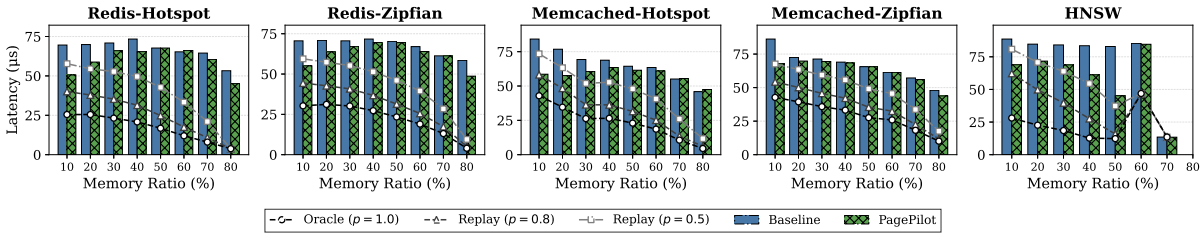


Fig. 8. **Average refault latency.** Bars show measured Baseline and ***PagePilot*** latency. Lines show latency estimates derived from the Oracle ($p = 1.0$) and the $p = 0.8$ and $p = 0.5$ replays using Baseline's backend-specific latencies. HNSW at 80% incurs no refaults.

Refault distribution analysis. Figure 7 shows that ***PagePilot*** generally increases the fast-backend refault ratio. The largest increase occurs for HNSW at the 50% memory ratio, from 9.8% to 65.5% (+55.7 percentage points). Under the most constrained YCSB configurations, the increases reach 26.8, 23.9, and 15.6 percentage points for Redis–Hotspot, Redis–Zipfian, and Memcached–Hotspot, respectively.

The full-foresight Oracle serves 58.1–100% of refaults from the fast backend. This result uses a global offline view: even for pages with similar reuse distances, their exact refault order determines how the limited slots should be scheduled. Reducing exact-information coverage moves many replay results away from the full Oracle. Nevertheless, ***PagePilot***, using only observed reuse history, approaches the $p = 0.5$ replay in several HNSW configurations.

Average refault latency. Figure 8 shows that ***PagePilot*** reduces mean refault latency by 10.8% on geometric average and by up to 45.5%. For HNSW at 50%, the increase in fast-backend refault ratio reduces latency from 82.9 to 45.2 μs. The replayed latency moves away from the full-foresight result as less exact future information is revealed; when all HNSW refaults use the fast backend at 70%, the measured and Oracle latencies converge. Together, the higher fast-backend refault ratio and lower refault latency explain the application performance in Figure 6.

4.3 Performance Improvement Breakdown

To isolate the performance impact of the *Offloading Router* and *Backend Migrator*, we compare three configurations: Baseline, *Router*, which enables only the Offloading Router, and the complete ***PagePilot*** system. For brevity, we refer to the two components as the *Router* and *Migrator* throughout this section and in the figures. The two components can be enabled independently. For each workload and memory configuration, ***PagePilot*** uses the component mode specified in Figure 9, which remains fixed throughout the measured run. At configurations where routing becomes overly restrictive, the Router operates in pass-through mode, allowing the Migrator to act on fast-resident pages.

Figure 9 shows the throughput contribution of the two components. Under high memory pressure, Router-only typically captures most of the performance benefit of ***PagePilot***. In this regime, pages enter swap frequently and fast-backend capacity is highly contended. Using historical refault distance to improve the initial backend decision prevents weakly reused pages from occupying the fast backend and preserves its limited capacity for pages that are more likely to refault soon. For example, at the 10% memory ratio, Router-only increases Memcached–Zipfian throughput by 64.7%. At the 50% memory ratio for HNSW, Router-only improves throughput by 65.2%.

The Migrator uses reuse information accumulated after the initial placement to correct persistent backend misplacements. By relocating pages that remain on the fast backend without refaulting for a long time, it releases fast-backend capacity for subsequently evicted pages with greater reuse value. This correction further improves

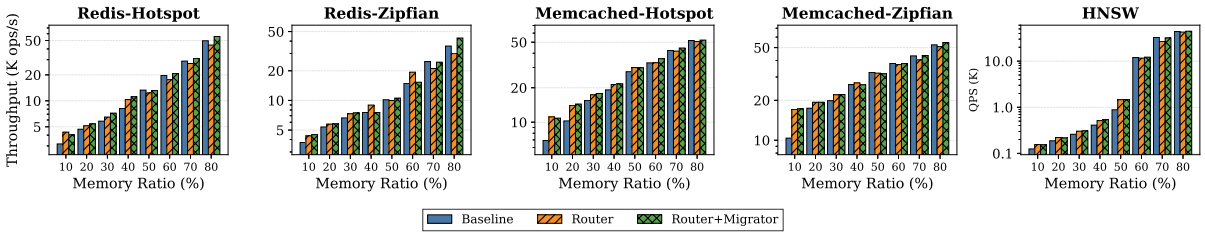


Fig. 9. **Throughput breakdown.** Absolute throughput of Baseline, Router-only, and Router+Migrator(full ***PagePilot***). At 50–80% for Redis–Hotspot and Memcached–Zipfian, at 50% and 70–80% for Redis–Zipfian, and at 60–80% for Memcached–Hotspot and HNSW, the Router is configured in pass-through mode and does not block fast-backend admission; pages therefore follow the Baseline admission path, allowing the *Migrator* to operate on fast-resident pages.

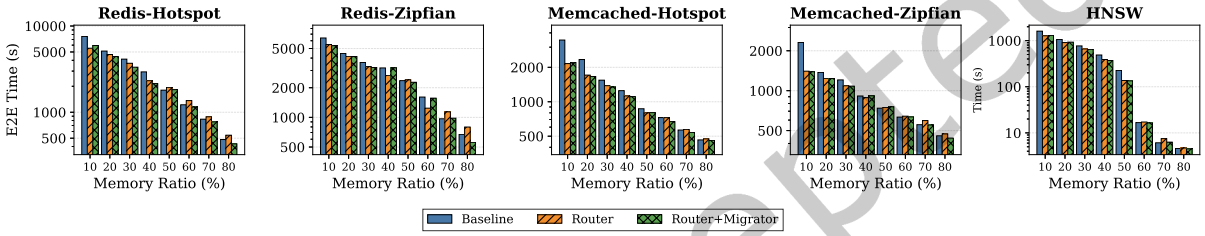


Fig. 10. **End-to-end completion-time breakdown.** End-to-end time for the configurations in Figure 9; fixed work makes the trend the inverse of throughput.

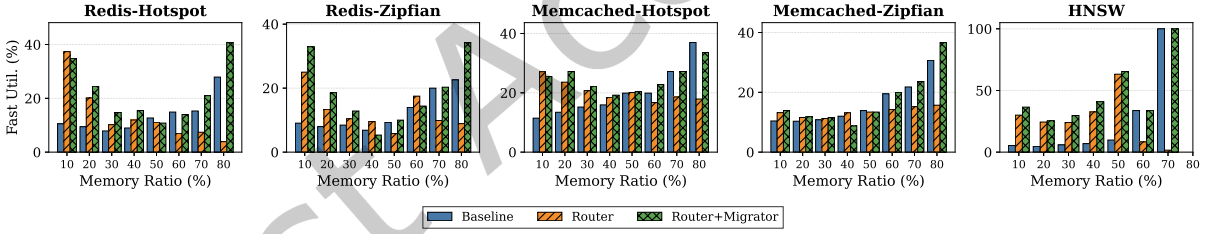


Fig. 11. **Fast-backend refault ratio breakdown.** Fast-served refault fraction for Baseline, Router-only, and ***PagePilot***, using the modes in Figure 9. HNSW at 80% incurs no refaults.

performance in several configurations. At the 40% memory ratio for Redis–Hotspot, Router-only increases throughput by 26.6%, while ***PagePilot*** extends the improvement to 36.8%. For HNSW at the same memory ratio, the corresponding improvement increases from 25.1% to 31.5%. These results show that the history available at eviction time is not always sufficient to avoid incorrect initial placement, and that correction based on subsequent behavior can further improve the effectiveness of the fast backend.

The marginal benefit of migration also depends on memory pressure. As more pages remain resident in DRAM, swap activity becomes less intensive and the Router is less likely to admit weakly reused pages into the fast backend. The number of persistent misplacements available for correction consequently decreases. Migration, however, still requires reading a page into DRAM and writing it to another backend. Moreover, the Migrator makes decisions based on previously observed behavior; a page moved to the slow backend may be accessed again

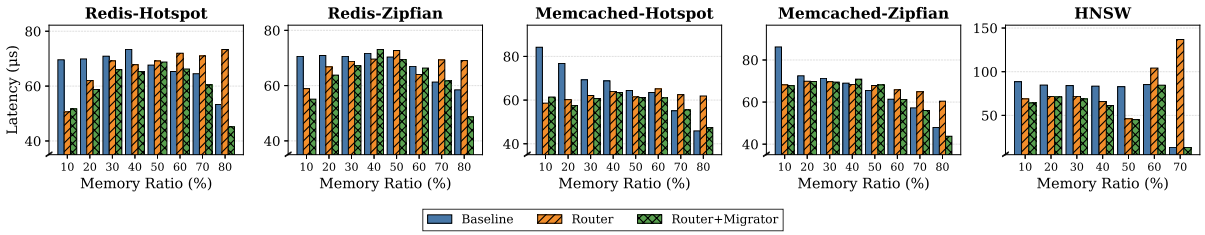


Fig. 12. **Average refault-latency breakdown.** Mean refault latency for Baseline, Router-only, and ***PagePilot***, using the modes in Figure 9. HNSW at 80% incurs no refaults.

shortly afterward, increasing the latency of its next refault. In some moderate-pressure configurations, these migration costs and occasional incorrect relocations outweigh the remaining correction opportunity. For example, at the 40% memory ratio for Redis–Zipfian, ***PagePilot*** selects Router-only because enabling the Migrator reduces performance.

At higher memory ratios, frequently and repeatedly accessed pages are more likely to remain in DRAM, while the pages selected by reclaim tend to have weaker future reuse. Applying restrictive admission to these pages offers little additional filtering benefit. Instead, it directly reduces their opportunity to enter the fast backend and can leave available fast capacity underutilized. In this regime, ***PagePilot*** disables the Router and retains only the Migrator.

Migrator-only does not reject pages at eviction time based on limited history. Pages are first placed according to the Baseline priority policy, allowing the fast backend to be filled normally. The Migrator then observes their subsequent behavior and relocates pages that remain on the fast backend without refaulting for an extended period. The default priority policy does not reconsider an already completed placement; consequently, pages that entered the fast backend early but are not accessed again can continue occupying slots and force later pages into the slow backend. By reclaiming these stale slots, the Migrator allows later evictions to use the fast backend.

This mode therefore combines permissive initial placement with delayed correction based on stronger runtime evidence. It avoids the underutilization caused by overly restrictive routing while retaining the ability to remove long-inactive fast-backend residents. The lower swap traffic at higher memory ratios also reduces contention between migration, demand refaults, and background reclaim. Migrator-only therefore remains beneficial across a broad range of moderate and high memory ratios. For example, at the 80% memory ratio for Redis–Hotspot, Router-only reduces throughput by 10.9% relative to Baseline, whereas ***PagePilot*** disables the Router and uses the Migrator to improve throughput by 12.3%. Similar behavior appears for Redis–Hotspot at 60–70% and across several high-memory Memcached configurations.

When the fast backend can already accommodate all pages that refault, neither routing nor migration provides additional placement opportunity. For example, at the 70% memory ratio for HNSW, Baseline already serves every refault from the fast backend. ***PagePilot*** therefore retains Baseline behavior rather than introducing unnecessary routing or migration.

Figure 10 shows the same trends in end-to-end completion time. When the Router or Migrator reduces demand refaults from the slow backend, application-visible fault stalls decrease accordingly. When a component is unsuitable for the current operating condition, disabling it avoids the corresponding routing or migration penalty.

Figures 11 and 12 connect these application-level effects to backend placement. Under high memory pressure, Router-only generally increases the fast-backend refault ratio and reduces average refault latency, while the Migrator further improves these metrics by correcting accumulated misplacements. At higher memory ratios,

restrictive routing may instead underutilize the fast backend; Migrator-only allows normal admission and removes only pages that remain unreferenced for a sufficiently long period.

For example, at the 80% memory ratio for Redis–Hotspot, the fast-backend refault ratio of Baseline, Router-only, and ***PagePilot*** are 27.9%, 3.9%, and 40.8%, respectively. Their corresponding average refault latencies are 53.3, 73.3, and 45.1 μs. This result shows that premature rejection by the Router substantially reduces the opportunity to use the fast backend, whereas delayed correction by the Migrator both avoids capacity underutilization and removes stale fast-backend residents.

Overall, the Router and Migrator provide two complementary forms of backend placement. The Router improves initial admission using reuse history when fast-backend contention is high, whereas the Migrator uses stronger runtime evidence to correct existing placement when initial classification becomes less effective or misplacements accumulate. Because the two components can be enabled independently, ***PagePilot*** can select between early filtering and delayed correction according to memory pressure and page-reuse behavior. It therefore delivers substantial performance improvements under high memory pressure while avoiding performance degradation as memory pressure decreases: when routing is no longer beneficial, ***PagePilot*** disables the Router and retains Migrator-only, or matches Baseline when neither component provides an advantage.

4.4 Parameter Settings

Table 3 summarizes the Router and Migrator thresholds used in the evaluation. The Router threshold N_r specifies the maximum average refault distance for admitting an evicted page to the fast backend, whereas the Migrator threshold N_m determines when a page residing on the fast backend becomes eligible for migration to the slow backend. Each table entry is reported as N_r/N_m.

Table 3. Router and Migrator thresholds used in the evaluation. Each entry reports N_r/N_m, where N_r and N_m denote the Router and Migrator thresholds, respectively. The table reports unscaled logical refault-distance values. The kernel implementation multiplies each value by 10^3 and uses fixed-point integer arithmetic to preserve the precision of average refault-distance calculations.

	Memory ratio							
Workload	10%	20%	30%	40%	50%	60%	70%	80%
Redis–Hotspot	30/30	15/15	8/10	8/8	10/15	15/25	30/30	35/35
Redis–Zipfian	60/80	10/25	5/7.5	10/10	10/25	25/25	25/25	35/35
Memcached–Hotspot	30/30	8/8	4/4	5/5	2/10	10/10	10/10	10/20
Memcached–Zipfian	40/50	20/20	15/15	5/15	10/10	10/10	15/15	20/20
HNSW	25/28	10/10	8/8	5/5	8/15	15/15	35/35	35/35

Overall, the thresholds tend to decrease first and then increase as the memory ratio grows. Because PagePilot measures refault distance in MGLRU generations, relaxing memory pressure reduces reclaim and slows generation advancement. A comparable reuse interval therefore corresponds to a smaller generation distance, and the Router and Migrator thresholds decrease accordingly.

As memory pressure decreases further, pages with short reuse intervals are increasingly likely to remain resident in DRAM and are rarely selected by reclaim. Pages that do enter swap consequently tend to have longer refault distances. Retaining a small threshold in this regime would cause the Router to reject too many pages from the fast backend, reducing its effective utilization. Similarly, the Migrator could relocate pages to the slow backend prematurely. The thresholds therefore increase again: the Router admits a broader range of evicted pages to improve fast-backend utilization, while the Migrator requires a longer period without a refault before

initiating migration. The exact turning point and magnitude depend on the reclaim rate and reuse distribution of each workload and therefore need not be identical across workloads.

The Linux kernel path avoids floating-point arithmetic. To improve the precision of average refault-distance calculations, ***PagePilot*** represents each threshold as a fixed-point integer scaled by 10^3. For example, a logical threshold of 30 in the table is stored as 30,000 in the kernel, while 7.5 is stored as 7,500. Performance is robust to small variations in these thresholds, so the evaluation uses simple, rounded values rather than relying on fine-grained parameter tuning.

4.5 Overhead

Methodology. We quantify ***PagePilot***'s overhead using a representative Redis–Hotspot run at the 20% memory ratio, where swap-out, refault, and migration activities are sufficiently frequent to exercise all major components. The experiment uses the same fixed routing and migration thresholds as the corresponding main-evaluation run. The one-time profiling procedure used to select these thresholds is therefore excluded from the steady-state breakdown.

We instrument disjoint ***PagePilot*** code regions using sampled `ktime` probes and report only the increments observed during the YCSB run phase. An empty probe costs 14 ns, which defines the measurement floor. The reported per-event costs include this instrumentation overhead and therefore conservatively upper-bound the direct execution cost. Two additional runs produce comparable per-event measurements. Table 4 summarizes the memory and runtime overheads.

Table 4. Per-component memory and runtime overheads of ***PagePilot***. Runtime measurements use Redis–Hotspot at the 20% memory ratio; WS denotes the 32 GiB application working set.

(a) Memory overhead

Metric	Folio association	Reuse-history record	Tracer mapping	Indirect remap table	Migrator queue	Total
Unit cost	8 B/page	12 B/record	No extra object	≤17 B/mapping	64-entry array	–
Peak footprint	∼500 MiB	95.0 MiB	0	≤1.2 MiB	∼1 KiB	≤596.2 MiB
Normalized footprint	0.195% of DRAM	0.29% of WS	0	<0.01% of WS	negligible	0.23% of DRAM
Core state per tracked page	8 B	12 B	0	–	–	20 B (0.488%)

(b) Runtime overhead

Metric	Tracer eviction	Router decision	Refault tracking	Lookup miss	Lookup hit	Migrator scan	Migration issuance
Events	71.86 M	51.63 M	72.37 M	41.03 M	68.4 K	39.84 M slots	71.7 K pages
Mean cost	264 ns	≤15 ns	58 ns	152.8 ns	469 ns	68.6 ns/slot	4.38 μs/page
Aggregate time	18.99 s	≤0.77 s	4.21 s	6.27 s	31 ms	2.73 s	0.31 s

Memory overhead. ***PagePilot*** introduces a fixed page-to-history association and a workload-dependent reuse-history record. The folio association incurs one 8-byte pointer per managed page. This field is provisioned for all managed physical pages rather than only the pages belonging to the evaluated workload. Our testbed contains 256 GiB of installed DRAM and approximately 250 GiB of managed physical memory; consequently, the system-wide association occupies approximately 500 MiB. This absolute footprint reflects the physical-memory capacity of the machine, whereas the normalized cost remains 8 bytes per 4 KiB page, or 0.195% of managed memory.

The reuse-history record consists of three 32-bit fields—the eviction timestamp, refault count, and average refault distance—for a total of 12 bytes per tracked page. At the measured peak of 8.30 million live records,

the history metadata occupies 95.0 MiB, or 0.29% of the 32 GiB working set. Together, the 8-byte association and 12-byte history record require 20 bytes per tracked 4 KiB page, corresponding to 0.488%. ***PagePilot***'s core per-page metadata therefore remains below 0.5% of the memory it represents.

The peak absolute footprint of all core ***PagePilot*** structures is at most 596.2 MiB on our testbed, corresponding to approximately 0.23% of its installed DRAM. The 0.23% platform-level footprint and the 0.488% per-tracked-page cost use different normalization bases: the former includes the association provisioned for every physical page, whereas the latter describes the metadata needed to track an individual page.

The Offloading Tracer does not allocate a separate mapping table. Instead, it reuses the swap-cache xarray slot that originally stores the vanilla shadow value, introducing no additional xarray entry or node. The Indirect Swap Entry Table reaches a peak of 70,943 mappings and consumes at most 1.2 MiB, including amortized xarray-node overhead. The fixed 64-entry candidate queue used by the migrator occupies only about 1 KiB.

Runtime overhead. Router decisions cost at most 15 ns. Tracer processing costs 264 ns per eviction, and refault-history updates cost 58 ns (0.09% of mean fault latency). An indirect-table miss costs 152.8 ns, or 0.28% of fast-backend fault latency; hits cost 469 ns but occur in only 0.17% of lookups. Migrator scanning costs 68.6 ns per slot, and issuing a relocation costs 4.38 μs per page. The 71,694 migrations move approximately 560 MiB of logical data. Across all instrumented regions, ***PagePilot*** consumes 33.8 s over a 4,926 s run, equivalent to at most 0.69% of one CPU core.

5 Discussion

5.1 Eviction–Offloading Co-design

Page eviction and page offloading have traditionally been optimized as two separate stages. The eviction policy determines which page should leave DRAM, whereas the offloading policy determines which backend should store the selected victim. ***PagePilot*** deliberately preserves this separation and does not replace the existing victim-selection policy. It therefore remains compatible with reclaim mechanisms such as MGLRU, DAMON-based reclaim, and iSwap [9, 17, 32].

Although the two stages can be implemented independently, their optimization objectives are closely related. The eventual cost of evicting a page depends not only on whether the page is reused, but also on where it is placed and how expensive a later refault will be. A page that repeatedly refaults shortly after eviction may be an undesirable victim even if it appears cold according to its recent DRAM-resident behavior. Conversely, a page that remains offloaded for a long time can be a reasonable eviction candidate, but placing it on a capacity-constrained fast backend prevents that capacity from serving pages with more imminent refaults. Decisions that are locally reasonable at either stage may therefore be suboptimal when their downstream effects are considered.

By preserving reuse history across eviction, backend residency, and refault, ***PagePilot*** exposes information that can be consumed by both stages. The current design uses this information to guide backend selection and correct persistent misplacements, but the same history can also support a deeper co-design. For example, a future eviction policy could reduce the eviction priority of pages that repeatedly experience short eviction–refault cycles, thereby accounting for the downstream cost of reclaiming them. In the opposite direction, eviction-side signals such as page generation, access frequency, reclaim confidence, or application priority could be incorporated into the offloading decision.

A joint policy could consequently consider both the immediate benefit of reclaiming a page and the expected cost of storing and later refaulting it from a particular backend. ***PagePilot*** does not require such a joint policy for its current operation. Instead, it provides the cross-stage history and placement interface needed to explore such policies without coupling the offloading framework to a particular reclaim algorithm. This separation also allows improvements in page eviction and backend placement to evolve independently while sharing a common view of page reuse behavior.

5.2 Concurrent Workloads and Cgroup-Scoped Adaptation

Concurrent workloads can exhibit substantially different swap and refault behavior. A routing threshold suitable for an interactive service may be too conservative or too permissive for a background analytics workload. ***PagePilot*** therefore supports cgroup-scoped policy state and parameter management, following the same resource-management boundary used by Linux memory reclaim. Workloads assigned to different memory cgroups can maintain and adjust their reuse-related parameters independently.

When multiple workloads are placed in the same memory cgroup, their per-page histories remain distinct, but they share one set of cgroup-level policy parameters. ***PagePilot*** treats these workloads as one combined memory-management domain and adjusts its policy according to their aggregate refault behavior. The resulting placement decisions optimize the fast-backend effectiveness of the cgroup as a whole, rather than attempting to optimize each constituent process independently. This behavior is suitable for multi-process services whose components share a common resource objective.

Workloads that require independent placement behavior can instead be assigned to separate cgroups. For example, an interactive service and a background batch job can use different cgroups so that their reuse patterns lead to separately adjusted routing parameters. This model also matches containerized deployments, where a cgroup commonly represents a service or resource-control domain rather than an individual process.

Within each policy domain, ***PagePilot*** uses PID-style feedback control to adapt the routing threshold to workload behavior and fast-device utilization. The controller adjusts the admission criterion according to the observed occupancy of the fast backend, allowing the system to admit more pages when capacity is available and to become more selective when that capacity becomes constrained. Cgroup-scoped parameter management allows workloads with different reuse distributions to use different operating points without requiring an offline model of each application.

This organization provides a flexible interpretation of a workload. A set of cooperating processes can be grouped together and optimized under one shared policy, while unrelated applications can be separated when independent adjustment is desired. ***PagePilot*** therefore follows the resource boundaries already used by the operating system instead of introducing a new application-identification mechanism into the swap path.

5.3 Policy Configurability and Applicability

PagePilot separates reuse-history tracking from backend-placement policy and exposes its main policy controls to user space. Administrators can enable or disable the Offloading Router and Backend Migrator, select automatic or manual threshold adjustment, and configure the parameters used by routing, migration, and occupancy feedback. Cgroup-scoped parameters are managed through the cgroup-v2 hierarchy, while system-level controls are exposed through the corresponding kernel file interfaces.

These interfaces provide bounded policy programmability. Users can adjust the behavior of ***PagePilot*** without changing or recompiling the kernel, while the actual decision logic remains a small and predictable in-kernel routine. ***PagePilot*** does not currently allow arbitrary user-provided callbacks or unrestricted classifiers to execute directly in the page-offloading path. Such an interface could provide additional flexibility, but it could also introduce unpredictable latency, unsafe behavior, or unstable placement decisions. A bounded interface is therefore better aligned with the latency and reliability requirements of kernel memory management.

The separation between mechanism and policy also permits future extensions without redesigning the history-tracking infrastructure. Additional predefined routing policies could consume the same reuse history, and administrators could select among them according to workload or deployment objectives. Similarly, richer policies may combine refault distance with application priorities or device-specific costs while retaining the same bounded execution model.

Reuse-aware offloading is most beneficial when three conditions hold. First, the volume of offloaded pages is large enough that the fast backend cannot accommodate all of them. Second, the offloaded pages exhibit diverse reuse behavior, such that some pages refault soon while others remain in swap for much longer periods. Third, the available backends expose a meaningful difference in refault latency. Under these conditions, allocation order alone cannot preserve fast-device capacity for the pages that benefit from it most.

The available optimization opportunity naturally decreases when swap activity is low or when the fast backend can accommodate most offloaded pages. This does not require ***PagePilot*** to impose a different placement policy in all deployments. Administrators can disable the Offloading Router and Backend Migrator when reuse-aware placement is unnecessary, or selectively enable the Router without background correction when the additional migration is not justified. The experiments under moderate and low memory pressure quantify the resulting performance and overhead as the placement opportunity diminishes.

The current prototype focuses on systems with two effective backend classes, matching common deployments such as compressed memory combined with an SSD. Within this scope, ***PagePilot*** provides a configurable mechanism for preserving fast-backend capacity and adapting placement to the observed reuse behavior of each workload domain.

6 Related Work

Refault events lie on the critical path of memory access, making the optimization of the swap system a topic of significant interest. Considerable efforts have been directed toward improving the accuracy and efficiency of page reclamation, thereby alleviating the overhead of refaults. For example, the recently introduced MGLRU mechanism [9] in the Linux kernel organizes pages into multiple generations to improve reclaim accuracy and reduce scanning overhead. Other approaches, such as DAMON [17] and iSwap [32], use sampled access information to improve the identification of reclamation candidates. PageFlex [37] further delegates paging-policy decisions to user space through eBPF while retaining the underlying Linux paging mechanisms. These approaches improve or expose reclamation and prefetching decisions. In contrast, ***PagePilot*** is invoked after the existing reclamation policy has selected a victim page and determines only the backend to which that page is offloaded.

A closely related line of work manages resident pages across heterogeneous memory tiers through page migration. Linux automatic NUMA balancing supports a memory-tiering mode that samples page accesses through faults and promotes hot pages to faster memory [16]. HeMem [25] uses hardware performance counters to identify hot and cold pages and guide their placement across heterogeneous memory tiers. NOMAD [34] introduces transactional page migration and page shadowing to move migration off the application critical path and mitigate memory thrashing under fast-memory pressure. More recently, Tiered Memory Management Beyond Hotness [18] shows that access frequency alone may not accurately capture the performance impact of memory accesses. It introduces amortized offcore latency, which accounts for both memory-access latency and memory-level parallelism, and uses this metric to guide profile-based object placement and regulate unnecessary page migrations. These systems operate on memory-resident pages and optimize placement among byte-addressable memory tiers. ***PagePilot***, instead, operates after eviction, when a nonresident page is represented by a swap entry. It complements resident-memory tiering by optimizing initial placement and corrective migration across heterogeneous swap backends.

Meanwhile, advancements in storage and transmission technologies, including SSD, NVMe, and RDMA, have significantly influenced the design of the swap subsystem. RDMA, in particular, offers promising bandwidth capabilities, enabling the development of disaggregated memory systems based on swap mechanisms. Numerous studies have explored extending memory systems with swap-based solutions, such as Infiniswap [7], FastSwap [1], Leap [21], and MemLiner [30]. These solutions demonstrate the continued potential of swap systems on modern hardware.

As more devices are integrated as swap backends, comparatively less research has focused on optimizing systems in which multiple heterogeneous backends are simultaneously available. Yoon et al. [38] consider a hybrid NVM-and-flash swapping architecture for smartphones and propose a storage-type- and partition-hotness-aware reclamation policy to mitigate swap-induced performance degradation. More recently, xDM [31] was proposed to maximize the aggregate processing bandwidth of multiple disaggregated-memory backends. It predicts backend bandwidth demand based on memory behavior and allocates backend devices among virtual machines at VM granularity.

In contrast, ***PagePilot*** focuses on page reuse behavior to improve swap-in performance. Unlike xDM, which allocates backend resources at the granularity of virtual machines, ***PagePilot*** selects the offloading backend for individual pages, allowing pages from the same workload to use multiple swap backends according to their expected refault behavior. Its Backend Migrator further corrects stale fast-backend placements without modifying the victim-selection policy. This page-granular design is complementary to existing hybrid-swap systems and remains compatible with containerized deployments and page-granular reclamation mechanisms.

7 Conclusion

This paper examines page offloading in modern systems that employ heterogeneous swap backends. We show that the default priority-based backend selection in Linux is insufficient in such environments and systematically leads to priority inversion in page placement, where fast devices are underutilized and frequently reused pages incur unnecessary latency on slow backends.

To address this problem, we propose ***PagePilot***, a reuse-aware offloading framework that augments the swap subsystem with lightweight reuse tracking, refault-distance–based placement, and background correction. Our implementation in the Linux kernel demonstrates that modest extensions to existing swap mechanisms can substantially improve both resource utilization and application performance under memory pressure.

Our evaluation across data-intensive and analytics workloads shows that ***PagePilot*** consistently improves throughput and reduces end-to-end latency, while increasing fast-device effectiveness. These results highlight page offloading as a critical, yet previously under-optimized, component of modern memory systems and motivate further reuse-aware designs for heterogeneous memory and storage hierarchies.

8 Acknowledgments

This work was supported in part by Natural Science Foundation of China (62672251,62372254), Innovative Development Joint Fund Key Projects of Shandong NSF (ZR2022LZH009, ZR2023LZH003), Beijing-Tianjin-Hebei Basic Research Cooperation Project of Hebei Natural Science Foundation under Grant F2024203115(24JCZXJC00160 in Tianjin).

References

[1] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 14, 16 pages. doi:10.1145/3342195.3387522

[2] Arch Linux Community. 2024. Swap. https://wiki.archlinux.org/title/Swap.

[3] ArchlinuxManual 2024. Retrieved April 13, 2024 from https://wiki.archlinux.org/title/Zram Archlinux - zram usage.

[4] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2019. *The datacenter as a computer: Designing warehouse-scale machines*. Springer Nature.

[5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. doi:10.1145/1807128.1807152

[6] Peter J. Denning. 1968. The Working Set Model for Program Behavior. *Commun. ACM* 11, 5 (1968), 323–333. doi:10.1145/363095.363141

[7] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[8] Junyeong Han, Sungeun Kim, Sungyoung Lee, Jaehwan Lee, and Sung Jo Kim. 2018. A Hybrid Swapping Scheme Based On Per-Process Reclaim for Performance Improvement of Android Smartphones (August 2018). *IEEE Access* 6 (2018), 56099–56108. doi:10.1109/ACCESS.2018.2872794

[9] Jonathan Corbet. 2022. Merging the multi-generational LRU. https://lwn.net/Articles/894859/.

[10] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: Re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 861–864. doi:10.1109/ICASSP.2011.5946540

[11] Saeed Kargar and Faisal Nawab. 2021. Extending the lifetime of NVM: challenges and opportunities. *Proc. VLDB Endow.* 14, 12 (July 2021), 3194–3197. doi:10.14778/3476311.3476406

[12] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 317–330. doi:10.1145/3297858.3304053

[13] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 17–34.

[14] Changlong Li, Liang Shi, Yu Liang, and Chun Jason Xue. 2020. SEAL: User Experience-Aware Two-Level Swap for Mobile Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39 (2020), 4102–4114. https://api.semanticscholar.org/CorpusID:226294212

[15] Xiancheng Lin, Jing Peng, Rongkai Liu, and Xiang Gao. 2024. Research on the CXL Memory. In *2024 IEEE 6th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, Vol. 6. 1428–1432. doi:10.1109/IMCEC59810.2024.10575852

[16] Linux Kernel Community. [n. d.]. Automatic NUMA Balancing and Memory Tiering. https://docs.kernel.org/admin-guide/sysctl/kernel.html#numa-balancing

[17] Linux Kernel Community. 2022. DAMON-based Reclamation. https://www.kernel.org/doc/html/latest/admin-guide/mm/damon/reclaim.html.

[18] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. 2025. Tiered Memory Management Beyond Hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 731–747. https://www.usenix.org/conference/osdi25/presentation/liu

[19] Jie Liu, Xi Wang, Jianbo Wu, Shuangyan Yang, Jie Ren, Bhanu Shankar, and Dong Li. 2024. Exploring and Evaluating Real-world CXL: Use Cases and System Adoption. arXiv:2405.14209 [cs.PF] https://arxiv.org/abs/2405.14209

[20] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836. doi:10.1109/TPAMI.2018.2889473

[21] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 843–857. https://www.usenix.org/conference/atc20/presentation/al-maruf

[22] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 742–755. doi:10.1145/3582016.3582063

[23] Sung-Kye Park. 2015. Technology Scaling Challenge and Future Prospects of DRAM and NAND Flash Memory. In *2015 IEEE International Memory Workshop (IMW)*. 1–4. doi:10.1109/IMW.2015.7150307

[24] pmhahn. 2019. VirtIO Memory Ballooning. https://pmhahn.github.io/virtio-balloon/.

[25] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 392–407. doi:10.1145/3477132.3483550

[26] Steven Rostedt. [n. d.]. ftrace – Function Tracer. https://docs.kernel.org/trace/ftrace.html.

[27] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 69–87. https://www.usenix.org/conference/osdi18/presentation/shan

[28] Shigeru Shiratake. 2020. Scaling and Performance Challenges of Future DRAM. *2020 IEEE International Memory Workshop (IMW)* (2020), 1–3. https://api.semanticscholar.org/CorpusID:219548290

[29] The Linux Kernel Documentation. [n. d.]. zswap. https://docs.kernel.org/admin-guide/mm/zswap.html.

[30] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 2022. MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 35–53. https://www.usenix.org/conference/osdi22/presentation/wang

[31] Jing Wang, Hanzhang Yang, Chao Li, Yiming Zhuansun, Wang Yuan, Cheng Xu, Xiaofeng Hou, Minyi Guo, Yang Hu, and Yaqian Zhao. 2024. Boosting Data Center Performance via Intelligently Managed Multi-backend Disaggregated Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 37, 18 pages. doi:10.1109/SC41406.2024.00043

[32] Zhuohao Wang, Lei Liu, and Limin Xiao. 2024. iSwap: A New Memory Page Swap Mechanism for Reducing Ineffective I/O Operations in Cloud Environments. *ACM Trans. Archit. Code Optim.* 21, 3, Article 47 (Sept. 2024), 24 pages. doi:10.1145/3653302

[33] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 609–621. doi:10.1145/3503222.3507731

[34] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 19–35. https://www.usenix.org/conference/osdi24/presentation/xiang

[35] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. 2024. FlexMem: Adaptive Page Profiling and Migration for Tiered Memory. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 817–833. https://www.usenix.org/conference/atc24/presentation/xu-dong

[36] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 331–345.

[37] Anil Yelam, Kan Wu, Zhiyuan Guo, Suli Yang, Rajath Shashidhara, Wei Xu, Stanko Novaković, Alex C. Snoeren, and Kimberly Keeton. 2025. PageFlex: Flexible and Efficient User-space Delegation of Linux Paging Policies with eBPF. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, Boston, MA, 291–306. https://www.usenix.org/conference/atc25/presentation/yelam

[38] Hyejung Yoon, Kyungwoon Cho, and Hyokyung Bahn. 2022. Storage Type and Hot Partition Aware Page Reclamation for NVM Swap in Smartphones. *Electronics* 11, 3 (2022). doi:10.3390/electronics11030386

[39] Xiao Zhu, Duo Liu, Kan Zhong, Jinting Ren, and Tao Li. 2017. SmartSwap: High-performance and user experience friendly swapping in mobile systems. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[40] Karl J. Åström and Richard M. Murray. 2021. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press.

Received 26 March 2026; revised 21 July 2026; accepted 21 August 2026