APEX: Access Pattern Driven Far-memory Prefetching via Hardware-assisted Profiling

Jinjiang Wang[1], Xiao Zhang[1*], Wendi Cheng[1], Shujie Han[1*], Xiaoling Shu[1], Xuqi Luo[2]
[1]*School of Computer Science, Northwestern Polytechnical University, Xi'an, 710029, China*
[2]*Electrical and Computer Engineering Department,*
College of Engineering, University of Alabama, United States
Email: {zhangxiao, shujiehan}@nwpu.edu.cn,
{jinjiangwang, chengwendi, shuxiaoling}@mail.nwpu.edu.cn,
xluo19@crimson.ua.edu

***Abstract*—Paging-based far-memory systems transparently increase node memory capacity by paging data to remote memory, requiring no application source-code changes. However, existing systems rely heavily on page-fault handlers to manage remote memory accesses. This design exposes a fundamental semantic gap between the operating system and the application: lacking visibility into application memory access patterns, the system initiates prefetching only after page faults occur, which leads to suboptimal predictions and sustained latency.**

To overcome these limitations, we propose APEX, which decouples pattern discovery from the page-fault path by using commodity PMU/PEBS sampling to learn access correlations asynchronously. APEX has three components: (1) PEBS-based sampling plus a counting Bloom filter to estimate page access frequency with low metadata overhead; (2) a compact correlation model (Top-K deltas) built for pages likely to fault; and (3) a fault-time decision module that either issues a targeted prefetch based on learned correlations or falls back to Linux readahead. We implement APEX in Linux 6.1.55 and evaluate it on multiple real-world workloads under varying local-memory ratios. Experimental results show that APEX outperforms state-of-the-art prefetching algorithms by up to 17%.

***Index Terms*—Far-memory, memory access pattern, prefetching strategy, kernel.**

I. INTRODUCTION

Memory-intensive workloads (e.g., ML training/inference and graph analytics) are pushing both memory capacity and bandwidth demands, making memory a primary bottleneck in modern datacenters. Data-centric applications now constitute a dominant portion of modern high-performance data centers [1]–[6]. Whether in hyperscale cloud infrastructures [7]–[9] or emerging edge environments [10], [11], there is an urgent need for expanded memory scales and efficient management mechanisms. Within this context, providing flexible and low-overhead memory management for the cross-layer services prevalent in cloud environments has become a critical research priority [12], [13].

Far-memory has emerged as a compelling paradigm to dismantle the memory wall of monolithic servers and significantly enhance resource utilization within data centers [14]–[22]. Existing far-memory mechanisms can be broadly categorized into three distinct approaches based on their implementation and abstraction layers. **1) Object-based far-memory systems** explicitly manage remote data units to optimize prefetching and access granularity [22] [23], [24], [25]. While these systems often achieve superior performance, they typically impose a significant integration burden by sacrificing application transparency, requiring developers to rebuild source code using specialized APIs. **2) Hardware-based far-memory systems** leverage emerging interconnect technologies, such as CXL, to provide hardware-level transparency for remote memory access [15], [26], [27]. Despite their seamless integration, these solutions introduce substantial architectural complexity, particularly regarding cache coherence protocols across extended bus interfaces [28], [29]. More critically, their adoption is currently hampered by the lack of widespread availability of commodity CXL-enabled hardware [30]. Furthermore, even with hardware support, sophisticated software-layer interventions remain necessary to mitigate microsecond-scale latencies exacerbated by inefficient concurrent remote accesses [31], [32]. **3) Paging-based far-memory systems** have received significant attention due to their immediate deployability [14], [17], [19], [33]–[36]. By leveraging the existing demand-paging and swap mechanism in the OS kernel, these systems enable a zero-effort transition for legacy applications. However, this transparency comes at the cost of heavy reliance on the synchronous page-fault handler. This design forces the substantial overhead of remote I/O operations and is tightly coupled with the application's critical path, leading to prohibitive performance degradation [30].

Ideally, far-memory systems strive to minimize remote memory accesses to mitigate the substantial latency penalties associated with page faults. To this end, contemporary frameworks such as Leap [19] and Fastswap [14] attempt to shorten the critical path by prefetching proximal pages into the swap cache during fault handling. However, the efficacy of these mechanisms remains hampered by a fundamental semantic gap between the operating system and the application. Because paging-based systems learn only when faults occur, they face a sampling paradox: fewer faults reduce latency but also remove training signals; more faults provide signals but directly increase stalls [30]. This tension raises a critical research challenge: **how to achieve high-fidelity yet low-overhead tracking of application access patterns to drive efficient prefetching with software support?**

To address this challenge without relying on specialized hardware, we design a software-defined paging prefetcher in the operating system that provides both application transparency and high prefetching efficiency. We present APEX, an application-aware prefetching strategy that functions as an independent data plane to augment existing paging-based far-memory systems. APEX runs as an asynchronous kernel thread that continuously samples memory-access events via Intel processor event-based sampling (PEBS) with low overhead. It uses a counting Bloom filter (CBF) to estimate per-page access frequency and focuses correlation using Top-k method mining on pages that are likely to be evicted and later faulted. On a page fault, APEX consults learned correlations to issue targeted prefetches; otherwise, it falls back to Linux readahead. The contributions of APEX are summarized follow.

- APEX leverages PEBS to monitor memory access event of application with low overheads.
- APEX introduces CBF to estimate 4KB page access frequency and uses Top-k to learn memory access sequence.
- APEX adaptively chooses to prefetch page from far-memory systems based on the learned-rule sequence or linux default prefetching strategies.
- Experimental results show that APEX outperforms state-of-the-art prefetching algorithms by up to 17%.

The remainder of this paper is organized as follows: Section II presents the background and motivation of our work. The specifications on the design and implementation of our model are described in Section III. Section IV depicts the evaluation methodology and shows the experimental results. The related work is summarized in Section V and the discussion is illustrated in Section VI. At last, the paper is concluded in Section VII.

II. Background and Motivations

A. Background

1) Memory management in Linux: Modern hardware architectures typically manage memory at a 4KB page granularity. Upon each memory reference, the Memory Management Unit (MMU) consults the corresponding Page Table Entry (PTE); if the "present" bit is cleared, a page fault exception is raised, ceding control to the operating system. The kernel services this fault by retrieving the missing page from backing storage (e.g., a swap device). To amortize I/O latency, Linux employs a spatial prefetching strategy, speculatively loading a window of pages that are contiguous within either the swap partition or the virtual address space.

This demand-paging paradigm introduces two main categories of performance overhead. First, when the application accesses a page that the kernel failed to prefetch, a major page fault occurs, causing the process to block until synchronous I/O completes; this penalty often dominates execution time. Second, even when a page has already been prefetched into the swap cache, its first access still triggers a minor page fault. Although this avoids remote I/O, minor faults still incur non-negligible overheads, including mode switches, PT updates, and TLB effects, especially for latency-sensitive workloads.

2) Far memory systems and current prefetching algorithms: Far-memory has emerged as a transformative datacenter architecture for improving overall cluster resource utilization by harvesting underutilized memory from remote servers or dedicated memory-expansion boards. In a typical deployment, memory-constrained nodes access far memory pools over low-latency networks such as RDMA [14], [17], [37] or commodity TCP [22]. Applications may interface with far memory either through specialized APIs [22] [23], [24], [25] or through transparent integration mechanisms [14], [17], [19], [33]–[36], the latter of which typically leverages existing virtual memory subsystems to handle page-level swapping.

However, applications operating with a low local memory ratio via paging mechanisms impose stringent demands on system bandwidth. In extreme scenarios where local memory allocation is negligible, maintaining near-local performance necessitates network bandwidth that rivals local memory bus speeds—reaching up to 800 Gbps on modern platforms [38]. Consequently, even with an idealized prefetching mechanism capable of masking access latency, traditional storage media like HDDs remain impractical due to their sub-5 Gbps bandwidth [39]. In contrast, the evolution of network fabrics beyond 100 Gbps [40] has made it feasible to explore application execution under severe local memory pressure.

To mitigate the inherent access latency, most far-memory systems proactively fetch anticipated data into local caches. Existing frameworks [14], [19], [34] predominantly employ rule-based prefetching strategies that detect and follow linear or strided access patterns. Our analysis suggests that such approaches offer limited benefits, as their performance gains are confined to applications exhibiting highly predictable and regular memory access sequences.

3) Processor event-based sampling: Tracing memory accesses is instrumental in characterizing application runtime behavior. Traditional page-table scanning techniques, such as the Accessed-bit based LRU, identify active pages by periodically inspecting reference bits. However, these software-defined approaches suffer from several fundamental limitations. First, they incur non-trivial performance overheads by triggering frequent page faults or costly TLB shootdowns to maintain consistency. Second, the scanning latency scales poorly with the expansion of memory capacity and the number of concurrent processes, exacerbating system contention. Most importantly, such methods offer limited observability; they can only detect whether a page was accessed within a coarse-grained interval, yet remain oblivious to whether the access was served by the hardware cache or main memory [41].

In contrast, hardware-based memory sampling offers a more profound insight into fine-grained access patterns [42], [43], [44], [45]. Modern processors feature hardware-level event sampling mechanisms, such as Intel's PEBS and AMD's Instruction-Based Sampling (IBS). By configuring specific hardware events (e.g., LLC misses) and sampling thresholds (e.g., once every 1000 events), these mechanisms can capture precise architectural state that its include process IDs and VA directly into dedicated hardware buffers. Unlike page table

scanning, PEBS provides high-fidelity address traces without intruding upon the critical path of the MMU or requiring exhaustive page table traversals [41], [45].

B. Motivations

In this subsection, we firstly explore an impact of I/O amplification on access latency in far-memory systems. Next, it provides a detailed characterization of the memory access behaviors of typical applications and illustrates the limitations of existing strategies. Finally, we discuss the challenges faced by both online and offline prefetching.

1) Impact of I/O Amplification on Access Latency: Paging-based far memory systems inherently suffer from I/O amplification when compared to object-based counterparts, as they fetch entire 4KB pages even for sub-page-sized data requests. While fine-grained access in object-based systems reduces bandwidth consumption, the extent to which object size dictates actual access latency warrants closer investigation.

We conduct a micro-benchmark based on two nodes running Ubuntu 22.04, and utilize libverbs to measure RDMA read/write latencies across varying object sizes on two nodes. As presented in Fig.1, fetching a 4KB page incurs only a marginal latency penalty that approximately 0.97 μs compared to a 64B cache-line-sized access. These results suggest that while paging-based prefetching amplifies bandwidth consumption, object granularity contributes only marginally to RDMA access latency; fixed network and software-stack overheads dominate.

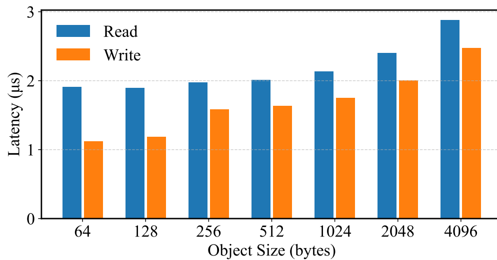

Fig. 1: Read/Write latencies of different objects via One-Sided RDMA (μs).

2) Memory access pattern of workloads: A wide spectrum of data-intensive applications exhibits diverse memory access patterns that defy capture by simplistic heuristic rules. To quantitatively analyze these behaviors, we employ the Intel Pin tool to intercept VA during READ and WRITE operations. We adopt a sampling granularity of one recorded address per million memory instructions, providing a high-fidelity trace of the application's memory footprint.

Fig.2 presents memory access of workloads, where the x-axis represents execution time and the y-axis denotes the VA space (mapped to unique IDs). As illustrated: (a) the 3D Poisson equation solver exhibits distinct strided access trajectories; (b) the Kmeans represents a class of iterative machine learning workloads characterized by predictable linear scanning and high spatial locality; (c) the NumPy-based matrix multiplication maintains highly contiguous scanning patterns; and (d) the Stream reveals a complex, interleaved access profile. Although these workloads exhibit structure in their address traces, the structure is often phase-dependent, interleaved, or data-dependent, making simple linear/stride heuristics insufficient. This inherent complexity necessitates a more complex to bridge the gap between raw access traces and effective prefetching.

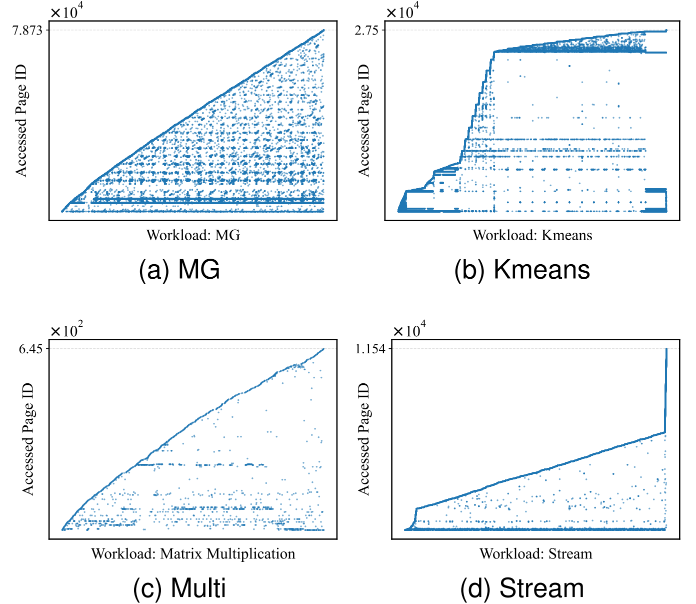

Fig. 2: Memory access pattern (READ/WRITE) by Intel Pin tool under four workloads.

Challenge#1: How can memory access patterns be captured online for the many applications in data centers?

3) Limitations of current strategies: We conduct our evaluation on a far-memory system based on Fastswap [14], deployed on Ubuntu 22.04 with Linux kernel 6.1.55. To establish a robust performance baseline, we compared our approach against standard Linux prefetching policies VMA [46] and Cluster and implemented the core mechanisms of Leap [19] with a fixed window size of 8.

To assess system resilience under varying degrees of memory pressure, we executed a diverse suite of workloads across a range of local memory ratios. For instance, a ratio of 0.9 allocates 10% of the working set to far memory, effectively triggering far page-faults and subsequent prefetching operations. This controlled configuration allows us to observe how different prefetching strategies mitigate the latency overhead of far-memory access as the local memory availability decreases. As the fraction of memory allocated to far nodes increases (i.e., the local memory ratio drops from 0.9 to 0.5), the OS tries to evict a larger volume of active pages to far memory. Consequently, subsequent accesses to these remote pages trigger a heightened frequency of page faults, necessitating expensive fault-in operations that stall the CPU. To show current performance gap among these strategies, we quantify the overall performance across the evaluated workloads as illustrated in Fig.3.

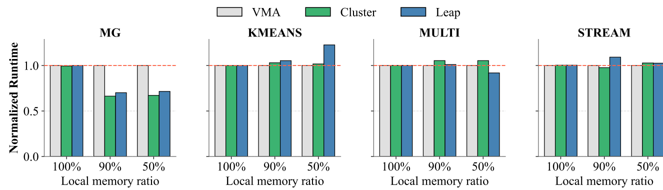

Fig. 3: Performance comparison of prefetching strategies across different applications under varying memory ratios.

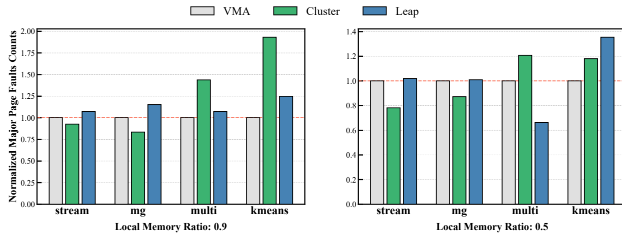

Fig. 4: The number of major page fault analysis of prefetching strategies across workloads under different memory ratios.

To further characterize performance loss, we quantify the number of major page faults across the evaluated workloads as presented in Fig.4. We employ the default VMA prefetching strategy as the experimental baseline. For the Multi workload, which is characterized by high sequentiality, Leap significantly reduces execution time compared to both VMA and Cluster as the memory offloading ratio increases. This performance gain is further elucidated by Fig4, which shows that Leap effectively suppresses major page faults in the Multi workload as the local memory ratio scales down from 90% to 50%. These findings provide a consistent cross-validation between throughput improvements and fault reduction.

However, Leap yields marginal gains or even performance degradation in other workloads such as Kmeans, MGP, and Stream. This discrepancy primarily stems from the inherent inability of existing prefetchers to capture non-linear memory access dynamics. Fundamentally, current prefetching strategies are hindered by a profound semantic gap between the operating system and the application. Being agnostic to the application's internal access sequence, the OS-level prefetcher is restricted to a reactive learning process triggered solely by page faults. Consequently, the algorithm is forced to perform inference on a sparse and fragmented dataset of access history, ultimately leading to suboptimal prefetching decisions.

Challenge#2: How to design efficient prefetching strategy based on memory access patterns in the far-memory systems?

4) Offline VS Online prefetching analysis: Efficient prefetching necessitates the capture of memory access patterns with both high fidelity and minimal latency. Traditional offline profiling via dynamic binary instrumentation (DBI) tools, such as Intel Pin, provides exhaustive memory traces with instruction-level precision. However, these tools impose a prohibitive runtime overhead that its often exceed a $100\times$ slowdown, which makes them unsuitable for tracking live workload dynamics. Furthermore, the act of prefetching inherently alters memory residency and I/O contention; thus, rules derived from offline traces often suffer from profiling bias, failing to reflect the shifted execution landscape.

Alternative software-based approaches, such as periodic PTE access-bit scanning, remain technically insufficient. These methods provide only coarse-grained touched indicators, failing to distinguish between cache hits and genuine DRAM-bound requests. On the opposite end of the spectrum, specialized hardware-assisted techniques (e.g., FPGA-based sniffers) offer low-latency monitoring but rely on non-standard components, which severely limits their deployment scalability in commodity cloud environments [47].

Consequently, online profiling leveraging PMU or PEBS emerges as a compelling alternative for capturing application phases in real-time. Despite inherent constraints such as sampling distortion and address sparsity, the negligible performance impact and hardware ubiquity of PEBS enable non-intrusive, continuous monitoring of DRAM accesses within the kernel. This provides a compromising foundation for an online, adaptive prefetching system.

Challenge#3: How to design prefetching strategy based on PEBS in the far-memory systems?

Taken together, these challenges motivates us to design an online, application memory access-aware performance prefetcher for far-memory systems.

III. Design of APEX

A. Overview

To effectively enhance prefetching efficiency and mitigate performance degradation in far-memory systems, we propose APEX, an online prefetching framework driven by application-level memory access insights. The design of APEX is guided by the following goals:

- **Application transparency.** APEX must operate within the OS kernel to transparently monitor memory behaviors without modifications to the application stack. It should remain agnostic to workload types and RSS to ensure broad applicability across diverse cloud environments.
- **Real-time adaptive decision-making.** Rather than relying on rigid offline sampling, APEX must perform online analysis of memory traces. This allows the system to make autonomous, real-time prefetching decisions that adapt to dynamic phase shifts in execution.
- **Minimal metadata overhead.** To maintain scalability as the application's memory footprint grows, APEX must avoid attaching extensive per-page metadata. The design prioritizes space-efficient data structures to prevent memory bloat and ensure that the benefits of far-memory expansion are not offset by auxiliary storage costs.

APEX is implemented as a kernel daemon bound to the application's lifecycle, which synchronizes its monitoring and processing activities with the process's execution. The core objective of APEX is to proactively prepare candidates set of memory pages for prefetching.

To address **C#1 & C#3**, APEX initiates with high-fidelity access tracking. APEX firstly leverages Intel PEBS technology

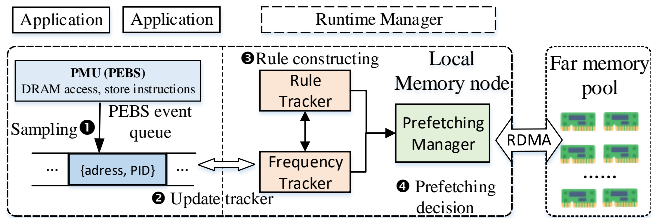

Fig. 5: Overview of APEX.

to capture precise VA associated with loading action of memory events ❶. Secondly, to manage stream of memory access address efficiently, we employ a CBF as frequency tracker. Given its minimal memory footprint and low computational complexity, the CBF allows us to map memory addresses to hash values and implement a periodic cooling mechanism to distinguish between hot and cold pages ❷. To address **C#2**, APEX build memory access rule tracker based on ❶ & ❷. APEX selectively chooses those infrequent accessed page (cold pages), then subsequently constructs temporal correlation metadata to map potential prefetch rule❸. Finally, upon a page fault, APEX executes prefetching management, APEX queries these correlations rule based on the faulting address to execute a tailored prefetching decision❹. Notably, the entire pipeline that memory access tracking, frequency tracker, rule tracker and prefetching management is executed asynchronously in the background. By decoupling these operations from the application's critical path, APEX ensures performance gains without introducing additional latency overhead.

B. Sampling memory access using PEBS

Inspired by prior tiered-memory work [41], [45], [48], [49], APEX uses Intel PEBS to sample memory-access events and estimate page-level access frequency at finer granularity than access-bit scanning–based LRU variants (e.g., LRU/MGLRU). We configure PEBS to sample MEM_INST_RETIRED.ALL_LOADS to capture a broader view of load traffic, and we later filter samples at page granularity. We focus on load events because far-memory paging is triggered by demand reads on missing pages, and load-miss traces better reflect the order in which pages become performance-critical. Stores are indirectly captured in many cases via read-for-ownership/read-to-share transactions and would add noise with limited benefit for predicting page-fault sequences.

We record virtual addresses (VAs) from PEBS samples and map them to 4KB page numbers for correlation mining. VAs are stable for a given mapping interval, and using VAs avoids requiring additional hardware support to directly capture physical addresses in kernel-space. Given that intercepting every memory event is computationally prohibitive and the Linux kernel autonomously throttles high-frequency sampling, we implement an adaptive sampling controller to bound profiling overhead. The sampling thread periodically measures its CPU utilization and updates exponential moving average (EMA) of CPU utilization. If utilization exceeds a target budget, APEX increases the PEBS sampling period via PERF_EVENT_IOC_PERIOD [41]; otherwise it decreases the period to improve fidelity. In our implementation, the initial period is set to 199 cycles and the control target is to keep the profiling overhead within 1% of one core. This feedback loop strikes an optimal balance between sampling fidelity and performance interference. Empirical evaluation shows that this kernel thread introduces a negligible CPU overhead of only 1.52% on a single core.

C. CBF-based frequency tracker

CBF requirement analysis. PEBS produces a sampled (not exhaustive) VA stream that is noisy and phase-dependent. Therefore, APEX avoids maintaining per-page exact counters and instead uses a compact approximate structure to track relative access intensity over time. While higher sampling interval inevitably increase CPU interrupts for PEBS buffer processing, our design minimizes per-sample overhead by deferring complex hot/cold page identification. Compared to maintaining exhaustive access metadata within page structures, CBF offers a cache-friendly and space-efficient alternative [49]. CBF matches PEBS sampling in that both are approximate: PEBS provides a probabilistic subset of accesses, and CBF further compresses these samples into approximate per-page frequency estimates with bounded memory, thereby fulfilling our design objective of minimal memory footprint. Thus, we employ the CBF as the core filtering mechanism to track access patterns and identify cold pages.

Upon a page fault occuring in the far-memory systems, the OS intercepts the faulting VA and proactively fetches 4KB pages from far memory to amortize access latency. To maintain consistency with this granularity, we canonicalize each sampled VA to its 4KB page number (VPN/PFN-equivalent at page granularity) because paging and swapcache operate at 4KB, and fault handling/prefetching decisions are ultimately made per page. By leveraging the probabilistic efficiency and low computational complexity of the CBF, we effectively map the distribution of VA. This approach mitigates data sparsity within the raw sampling traces, enabling the system to extract stable and deterministic memory access rules for subsequent prefetching.

Example of CBF. A CBF is functionally defined by a set of H independent hash functions and a counter array of size M. Our implementation supports two fundamental primitives: **GET** and **INCREMENT**. The **GET** operation computes H indices derived from the hash functions and retrieves the minimum value among the corresponding counters as the frequency estimate. Conversely, the **INCREMENT** operation identifies these H indices but selectively updates only the counters holding the minimum value to mitigate overestimation. Furthermore, to prevent stale hotness from biasing cold page identification and to maintain high counting fidelity, we introduce periodic **DECREMENT** mechanism by right-shifting all counters, effectively halving historical counts to prioritize recent phases and prevent stale hotness from dominating cold-page selection.

Fig.6 provides a example of CBF with $H = 3$ and $M = 8$. As depicted in Fig.6(a), a **GET** request for Page A initially returns 2; a subsequent **INCREMENT** on Page A updates the counters at indices 0 and 4 to a value of 3. Similarly, for Page B, the **GET** operation returns a frequency of 6, which is then incremented to 7, as reflected in subsequent queries. As shown in the Fig.6(b), the **DECREMENT** operation (Cooling) shifts all counter values one unit to the right. Consequently, a re-query for Page A yields a decayed frequency of 1, demonstrating the system's ability to prioritize recent access intensity over long-term history.

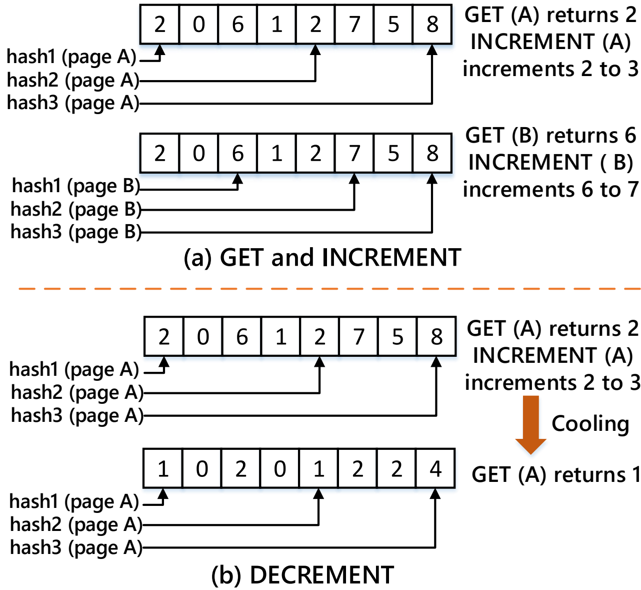

Fig. 6: Example of CBF.

As presented in the Fig.6, the access count of a specific page may be inadvertently overwritten by others due to the inherent hash collisions within the CBF. We define this phenomenon as tracking error. To strike a judicious balance between tracking fidelity and memory overhead, we dimension the CBF size m using the established probabilistic model [50]:

$$r = \frac{-k}{\ln(1 - \exp(\ln(p)/k))}, \quad m = \lceil n \cdot r \rceil$$

where p denotes the target probability of tracking errors, k represents the number of hash functions, and m is the total number of counters in the CBF. In our implementation, we configured $k = 5$ and set m to be proportional to the total number of swappable memory pages. This specific parameter combination proved to be robust across all evaluated workloads, effectively mitigating aliasing effects while maintaining a negligible memory footprint.

CBF-based cold page identification. In current far memory systems, OS typically employs 2Q-LRU or MGLRU scanning mechanisms to identify cold pages, maintaining the local memory watermarks while prioritizing hot pages in the local tier. However, subsequent accesses to these swapped-out pages trigger expensive page faults. Moreover, performing fine-grained tracking across the entire memory footprint introduces prohibitive CPU overhead. To mitigate this, APEX selectively models correlations for pages that appear cold (i.e., low sampled frequency) to bound metadata and computation; it still samples accesses globally via PEBS.

Upon receiving a VA via PEBS sampling, we extract its 4KB page frame number (PFN) and perform **GET** and **INCREMENT** operations within a CBF. The CBF then yields a minimum hash value for the PFN, which is evaluated against a predefined heat threshold to determine whether the page warrants a prefetching rule. To simplify the architectural design, APEX adopts a baseline derived from half the process's memory footprint (quantified by the number of 2MB huge pages). A page is categorized as cold if its corresponding hash value in the CBF falls below this baseline. Furthermore, to maintain the temporal relevance and frequency accuracy of the CBF, we implement a periodic aging mechanism. We trigger **DECREMENT** when the number of processed samples reaches a footprint-proportional budget, ensuring aging frequency scales with address-space size and keeps the CBF responsive to phase changes.

D. CBF+top-k rule-mining tracker

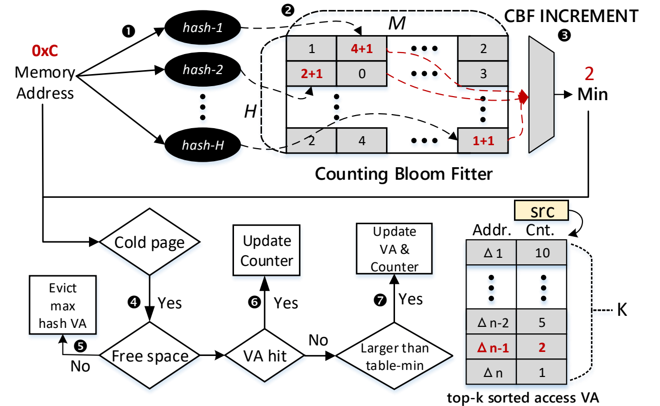

Fig. 7: Workflow of CBF + topk rule constructing.

Exact per-page counting works offline but is impractical online: it requires large per-page state and long observation windows to reliably rank pages, which increases memory and CPU overhead under dynamic phases. These costs arise because collect-then-sorting requires (i) maintaining counts for most pages and (ii) periodically performing global ranking or heavy updates both expensive in the kernel [51].

To mitigate these bottlenecks, APEX avoids global ranking. It first uses the CBF to identify low-frequency (cold) pages in the PEBS stream (❶-❹), and only for these pages it mines the next K page accesses to build compact correlation rules (❺-❼). This filter-then-mining design bounds both computation and metadata while focusing on pages that are likely to be evicted and later faulted. This selective profiling strategy effectively bypasses the high computational complexity of global sorting, enabling real-time, low-overhead pattern discovery.

Preprocessing for Top-K rule generation. Upon intercepting a VA via PEBS sampling, for each sampled VA,

APEX maps it to the current page P_t. If P_t is classified as cold, we treat it as a trigger page src. We then look forward in time using the per-CPU recent-access buffer (i.e., the accesses recorded after src was observed) to collect up to K successor pages as correlation candidates. To minimize metadata and storage overhead, we eschew recording absolute PFN sequences in favor of maintaining the relative distance (Δ) between candidate pages and the source page, defined as $\Delta = \text{page_id} - src$. Consequently, the transition from a source page to its successors is characterized by the tuple (src, Δ). This delta-based representation effectively compresses the access footprint while preserving the underlying spatial-temporal access patterns.

Construction of Top-K Access Rules. To characterize the predominant successor behaviors of memory accesses within a constrained footprint, APEX maintains a local Top-K statistical state for each identified cold source page (src). This state tracks the K most frequent access offsets (Δ) alongside their respective occurrence weights (counts). We employ the Space-Saving algorithm [52] that a robust frequent-item approximation technique to manage these Top-K updates under strict memory budget. Specifically, if an incoming Δ already exists in the Top-K set, its associated count is incremented. If the set is not yet saturated, the new entry is inserted directly. Upon reaching the capacity K, we adopt the standard Space-Saving replacement strategy: the entry with the minimum count is evicted and replaced by the current Δ, with its count initialized according to the approximation increment rules. Since our rule construction is selectively applied to cold pages, the Top-K modeling naturally gravitates toward potential page-fault trajectories. This targeted approach effectively mitigates redundant modeling of hot pages and prevents cache pollution. Furthermore, to adapt to evolving application phases, the existing rule for a given src is updated or overwritten upon its subsequent re-occurrence as a cold page source.

Storage of Top-K Access Rules. To manage global prefetching rules, APEX employs a hashed bucket structure organized in the format of triggering Page $\rightarrow$ Top-K(Δ). Each bucket comprises a fixed number of state slots, each maintaining the statistical Top-K deltas associated with a specific triggering page. Concurrent consistency during updates is ensured via bucket-level synchronization primitives, while lookups utilize the page fault address (pf) as a key to locate the corresponding bucket. Upon a successful hit, the system extracts the Top-K deltas, reconstructs the candidate prefetch addresses as $pf + \Delta$, and generates a bounded prefetch list. If no rule exists for pf, APEX falls back to Linux's default readahead (VMA policy), ensuring safe degradation to baseline behavior when confidence is low.

Given the finite capacity of each bucket, a robust eviction policy is essential when allocating slots for new cold pages. Because table capacity is limited, we need an eviction policy for src entries. Our goal is to prioritize entries that are most likely to fault in the future. When a bucket is full, we evict entries whose trigger pages appear less fault-prone under current sampling. In practice, pages with higher sampled frequency are more likely to stay resident in local memory and thus less likely to fault; therefore, we preferentially evict hotter trigger pages to keep capacity for colder, eviction-prone pages.

In summary, these mechanisms enable the rule table to adaptively self-configure within a constrained memory footprint. By intentionally bypassing hot pages and prioritizing the residency of cold page metadata, APEX provides a highly targeted candidate set for prefetching precisely when a page fault occurs.

E. Prefetching management

After producing candidates from the faulting page, APEX avoids issuing swap-ins indiscriminately. It validates candidates in the fault path to reduce unnecessary network traffic and to prevent swapcache pollution. We issue a swap-in only if the candidate page is (i) non-present in the page table and (ii) absent from swapcache. This check also suppresses redundant prefetches under concurrent faults where another thread may have already brought the page in. Furthermore, APEX introduces a density-based threshold to govern the activation of its prefetching logic. The prefetching strategy is invoked only when the number of valid candidates exceeds a majority threshold (i.e., $> 50\%$); otherwise, the system gracefully falls back to the default Linux readahead mechanism to ensure robustness under low-confidence scenarios. Regarding cache eviction, we maintain strict consistency with the upstream Linux reclamation policy to ensure seamless integration and predictable memory pressure management.

F. APEX works in Linux kernel

APEX operates in the Linux kernel as an asynchronous kernel thread. Algorithm1 illustrates the workflow of APEX operations. It continuously samples memory accesses of a target process via PEBS, canonicalizes sampled virtual addresses to 4KB pages, and maintains lightweight page-frequency estimates using a CBF with periodic aging to remain phase-aware. When a page is classified by the CBF as cold, APEX extracts its subsequent access sequence from a per-CPU recent-access ring buffer, encodes successors as relative page deltas (Δ), and applies the Space-Saving algorithm to maintain a Top-K correlation summary for that trigger page. These rules are stored in a capacity-bounded rule table; when necessary, APEX evicts entries corresponding to hotter trigger pages to preserve space for fault-prone pages. Upon a page fault, the kernel consults the rule table using the faulting page as the key, generates candidate pages, and validates them before issuing swap-ins. If the number of valid candidates exceeds a confidence threshold, APEX performs targeted prefetching; otherwise, it safely falls back to the default Linux readahead policy (e.g., VMA/Cluster).

Overall, these components of APEX operate as independent functional modules within the kernel and are organized as a pipelined workflow on a single kernel thread. In practice, APEX build prefetching candidates with low overhead, and it incurs only 1.52% of single CPU core and introduces an

Algorithm 1: APEX's Working Procedure

```
Input: Target application PID
   // Stage 1: Hardware-assisted PEBS Scanning
1  for each sampled event do
2      (addr, event) ← Pebs_Sample(PID, event);
   // Stage 2: CBF-based filter
3  for each sampled addr do
4      CBF_GET(addr);
5      CBF_INCREMENT(addr);
6      if N_scan == 1/2 × N_cgroup_pages then
7          CBF_DECREMENT();        // Halve values to
           maintain temporal locality
   // Stage 3: CBF-based Top-K mining
8  for each sampled pg do
9      if CBF_IS_COLD(pg) then
10         src ← pg;
11         S ← RECENT_SUCC(PID, src, K) // up to K
           successors after src;
12         foreach pg' ∈ S do
13             Δ ← pg' − src;
14             SPACE_SAVING_TOPK_UPDATE(src, Δ)
               // Space-Saving update on Top-K Δ
15         entry ← RULE_GET(src);
16         if entry exists then
17             entry.topk ← merge_topk(entry.topk,
               local_topk(src));
18         else
               // bounded rule table; evict if needed
19             if rule_table_full(bucket(src)) then
20                 victim =
                   arg max e ∈ bucket(src)CBF_HEAT(e.key);

                                // evict hotter trigger pages
                   delete(victim);
21             RULE_PUT(src, local_topk(src));
   // Stage 4: Prefetching management
22 for each page fault pf do
23     entry ← RULE_GET(pf);
24     if entry exists then
25         cand ← δ;
26         if |cand| > K/2 then
27             prefetch_swapin(cand)
28         fallback_readahead(pf);
```

additional memory footprint of 10.6 MB metadata overhead, which only accounts for about 0.1% RSS of workload (9 GB).

IV. Experiments

A. Experimental setup

a) Evaluation platform: We evaluate APEX on a two-node cluster interconnected via a 100 Gbps InfiniBand network. One node serves as the compute node, while the other acts as the far-memory pool. Each node has an Intel Xeon Gold 5218 (2 sockets × 16 cores, 2.3 GHz) and 64 GB DRAM. One node acts as the compute node and the other as the far-memory server. We implement and evaluate APEX on Linux 6.1.55 (Ubuntu 22.04), and we will release the patch for reproducibility.

b) Evaluation baselines: To evaluate the effectiveness of the proposed prefetching strategy, we use Fastswap [14] as the paging-based far-memory backend, and implement APEX as an alternative prefetching module in the swap fault path. We compare against (i) Linux VMA readahead, (ii) Cluster readahead, and (iii) Leap [19] implemented on the same backend.

c) Evaluation benchmarks: To evaluate the performance of APEX under diverse execution scenarios, we employ a broad suite of real-world workloads, as summarized in TableI. This selection encompasses not only classic compute-intensive kernels from standard benchmark suites such as K-means clustering, the Stream, and Multigrid PDE solvers, but also sophisticated graph processing algorithms executed on high performance frameworks, including Ligra. By incorporating both traditional memory-bound tasks and modern large-scale graph analytics, we ensure a comprehensive assessment of our system's adaptability to varying memory access patterns.

TABLE I: Benchmarks characteristics.

Benchmark	RSS	Description
Stream [53]	4GB	Stream for memory bandwidth
Multi [54]	2.4GB	Matrix multiplication on NumPy over matrices of floating points
Kmeans [55]	1.7GB	K-means clustering on sklearn
Quicksort [56]	8GB	Quicksort on c++ std
Lg-bfs [3]	9GB	Breadth-first search on ligra
MG [57]	3.4GB	Multi-grid PDE of NAS-benchmark

d) Evaluation metircs: Similar to prior work [19], we measure the performance of prefetching algorithms with four metircs: **Accuracy:** the ratio of total page hits and the total prefetched pages. **Coverage:** the ratio of the number of page hits from the prefetched pages and the number of remote requests plus the number of prefetch hits. **Normalized performance:** the baseline is the completion time when a benchmark uses local memory only. The normalized performance can be calculated as CT_{local}/CT_{system}, where CT_{system} and CT_{local} denote the completion time of far system and all local memory, respectively. **Normalized No. page faults:** This metric represents the total number of major page faults recorded across varying local memory ratios, normalized against the baseline Linux VMA policy.

e) Experimental scenarios: We evaluate the effectiveness of APEX to answer following questions:

- **Q1.** How does APEX improve the execution time of memory-intensive workloads compared with state-of-the-art prefetching strategies? (IV-B)
- **Q2.** What are the individual performance contributions of APEX's core components? (IV-C)
- **Q3.** Dose APEX reduce the number of page fault effectively in comparison with strategies? (IV-D)
- **Q4.** Dose APEX improve performance metircs accuracy and coverage under different local memory ratios in comparison with strategies? (IV-E & IV-F)

B. Overall performance

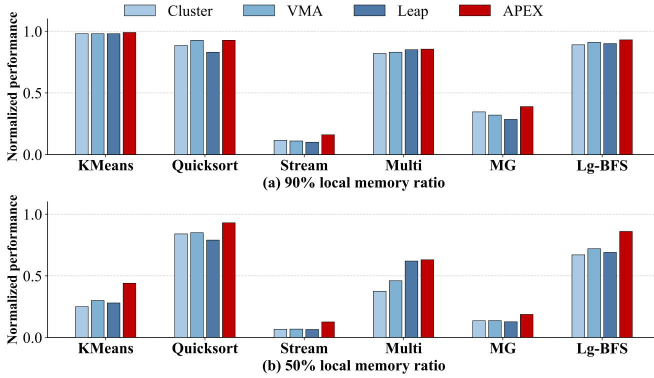

Fig. 8: Normalized performance.

As illustrated in Fig.8, APEX consistently outperforms VMA, Cluster, and Leap across the evaluated scenarios. As the remote memory ratio escalates from 10% to 50%, the intensified frequency of remote accesses leads to a performance degradation for all applications. This decline is primarily attributable to the inherent RDMA transmission latency and the non-negligible software-stack overhead involved in page fault handling.

It can be shown that APEX maintains better normalized performance in all workloads. This advantage stems from APEX's capability to capture memory access sequences online and proactively trace subsequent correlated access patterns upon a page fault trigger. At a 50% local-memory ratio, APEX improves performance over Leap by up to 1.17× at most across all workloads where access patterns are not purely sequential. For Multi workload, APEX's performance is slightly priority to Leap. This discrepancy arises because Multi exhibits highly sequential access characteristics, allowing Leap's majority-voting mechanism to accurately predict future page fault offsets. APEX leverages PEBS to build memory access sequence based on VMA prefetching algorithm. Nevertheless, by providing memory access information, APEX still delivers enhanced performance relative to these baselines.

C. Ablation analysis

To further validate the performance contribution of each component, we plan to conduct an ablation experiment on APEX with a local memory ratio of 50%. Following the execution flow of APEX, we define the components as follows: Component 1 (C1) represents the introduce of cold page identification; Component 2 (C2) represents the use of the Top-K method; and Component 3 (C3) represents the prefetching management.

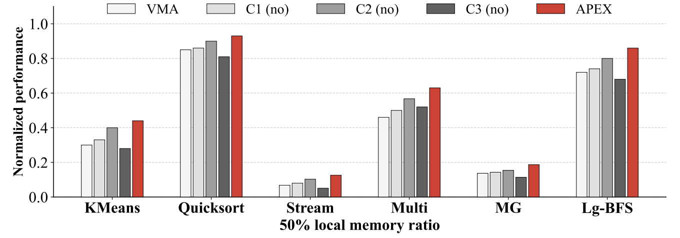

Fig. 9: Normalized ablation performance.

As shown in Fig.9, it is noteworthy that disabling prefetch management (C3) severely degrades prefetching efficiency and overall performance, yielding results even worse than the default Linux prefetcher, VMA. This is primarily because, although prefetch candidates are established based on application memory access information, performing state determination and filtering on the prefetching candidates set along the swap path leads to redundant page fetching and cache pollution. The scarcity of effective prefetch targets diminishes the performance benefits of prefetching. Secondly, when sampled pages are not distinguished as cold or hot (C1), the resulting performance gain is marginal. This is mainly because APEX fully stores the 4KB page frame numbers obtained from PEBS samples, making it oblivious to the application's access frequency and difficult to identify swap-out candidates. Specifically, frequently accessed memory pages remain resident in local memory node, whereas infrequently accessed pages are often swapped out to far memory. Finally, disabling the Top-K method (C2) has a minor impact on performance, approximately 3%. This can be explained by the fact that Top-K is used to identify the subsequent k frequently accessed pages for the current page, effectively capturing the nearby pages likely to be accessed next.

D. Overall normalized No. page faults

Fig.10 illustrates the number of major page fault across various workloads, normalized to the default Linux VMA-based policy as the baseline. It is evident that as the local memory ratio decreases (i.e., the far-memory proportion increases), a larger fraction of the working set is swapped to remote nodes. Subsequent CPU accesses to these remote pages inevitably trigger major page faults, necessitating RDMA-based data transfers.

The observed trends in page fault frequency are highly consistent with the normalized execution performance shown in Fig.8. Across the evaluated benchmarks with exception of the Multi workload, APEX significantly mitigates the occurrence of major page faults. Specifically, compared to Leap, which relies strictly on fault-triggered prefetching, APEX leverages its pattern-aware proactive mechanism to reduce

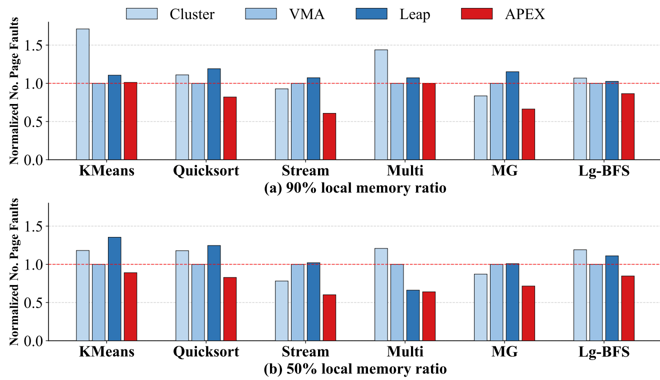

Fig. 10: Normalized No. major page faults (VMA-based). A larger value indicates a higher frequency of major page faults, whereas a smaller value indicates a lower frequency.

the number of major page faults by approximately 36% at most. This reduction underscores APEX's superior capability in accurately predicting future access streams and pre-staging data before the faulting path is even invoked.

E. Accuracy comparison

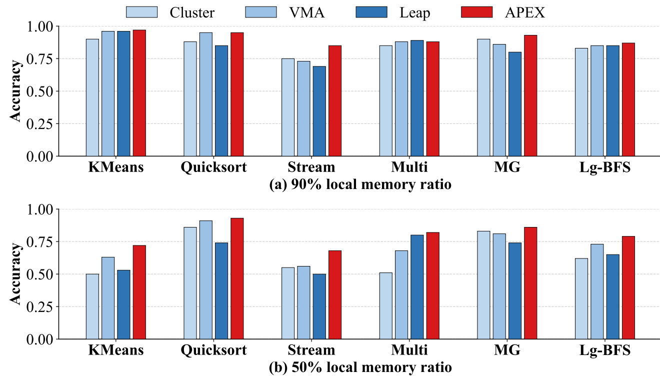

Fig. 11: Accuracy of prefetching strategies under different local memory ratio.

Fig.11 illustrates the prefetching precision of the default Linux VMA, Cluster, and Leap strategies across varying local memory ratios. Under configurations with abundant local memory (i.e., low remote memory ratios), the system encounters a minimal number of page faults. In such scenarios, all evaluated prefetchers consistently achieve precision exceeding 80%, indicating that the majority of prefetch requests are accurate, which significantly bolsters application performance.

However, as the remote memory pressure intensifies (e.g., at a 50% ratio), the OS triggers more page faults. Under these constrained conditions, APEX demonstrates better fidelity across most workloads with the exception of multi by maintaining high precision and reducing redundant network bandwidth consumption, effectively mitigating the impact of local memory pollution. Compared to Leap, APEX achieves an average precision improvement of approximately 16%. This gain underscores APEX's ability to navigate complex access patterns where traditional linear or cluster-based heuristics often falter.

F. Coverage comparison

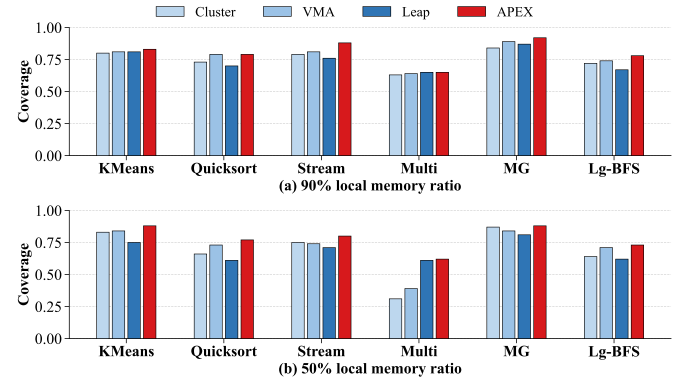

Fig. 12: Coverage of prefetching strategies under different local memory ratio.

Fig.12 presents the prefetching coverage across various prefetchers under different local memory ratios. In our design, a prefetching operation is triggered upon a page fault; a successful prefetch ensures that the target page is resident in the swapcache for subsequent accesses, thereby resulting in a direct hit. APEX leverages the application's recorded access sequences to proactively fetch subsequent memory pages into the swapcache.

While the improvement in coverage marginality remains relatively constrained in comparisons with the native Linux VMA and Cluster prefetchers, it can be illustrated that APEX provides insight to memory pattern recognition and with more precision. Notably, excluding the Multi workload, APEX achieves an average coverage enhancement of approximately 8% over Leap. This performance gain underscores APEX's ability to capture non-linear access correlations that conventional linear-stride-based prefetchers typically overlook. Coverage gains are inherently bounded because far-memory faults are dominated by a small set of recurring pages under each phase; once these pages are captured, additional prefetches tend to become redundant and are filtered by our validation logic, limiting further increases in coverage.

V. Related Work

Heuristic-based Prefetching Mechanisms. Existing literature on memory prefetching primarily revolves around heuristic strategies that anticipate future access patterns. One prominent category focuses on historical access correlations, where prefetching decisions are predicated on the program's recurring memory footprints within a single execution [19], [58]–[60]. Another distinct approach relies on the statistical measurement of working sets to guide directional prefetching [61], [62]. Recently, there has been a surge in learning-based methods, which employ machine learning models to infer future behaviors from complex historical access sequences

[63]–[65]. Furthermore, systems such as HoPP [30] integrate specialized hardware tracing tools (e.g., HMTT) to capture fine-grained memory traces, thereby enhancing prefetching precision. Distinguished from these works, APEX avoids the need for specialized tracing hardware. Instead, it capitalizes on ubiquitous platform features to implement a software-defined, pattern-driven heuristic that adaptively prefetches data in far-memory environments.

Object-based prefetching. Another line of research explores empowering application developers to explicitly provide prefetching hints [66]–[68] or granting fine-grained user-space control over memory paging [22], [25], [69], [70]. These approaches typically leverage program transparency to achieve memory expansion via remotable data structures, allowing developers to design bespoke prefetching strategies through specialized far-memory APIs. However, such solutions inherently rely on manual application-level intervention, where the resulting performance gains are strictly coupled with the developer's ability to accurately anticipate memory access patterns. In contrast, APEX provides an automated optimization framework that eliminates the need for manual code modifications, ensuring high performance without compromising developer productivity.

Paging-based prefetching in remote memory. Modern remote memory systems have explored diverse strategies to optimize swapping mechanisms and data prefetching. At the architectural level, Fastswap [14] achieves performance gains by streamlining the swap data path, while Hermit [34] further introduces an asynchronous, proactive evicting technique. Although these systems significantly enhance execution efficiency, they remain largely dependent on the default Linux prefetching policy. This native policy relies on simplistic, rigid prefetching rules that frequently fail to trigger effective data movement in diverse real-world workloads. Leap [19] represents a breakthrough by refactoring the Linux prefetcher; its core mechanism attempts to map application-level page access behaviors onto a set of predefined patterns, such as sequential or strided accesses. However, Leap struggles to maintain its efficacy when confronted with complex, non-deterministic memory access patterns that are heavily dependent on runtime dynamics, which elude static or template-based identification.

Tiered memory systems. Recent research has extensively explored strategies to overcome the DRAM capacity wall by integrating slower yet higher-capacity memory or storage tiers—such as compressed memory, Non-Volatile Memory (NVM), and NVMe SSDs. Representative systems, including HeMem [45], MEMTIS [41], and FlexMem [48], transparently offload main memory data to these secondary layers. To mitigate the performance degradation, these works predominantly leverage PEBS-based profiling to enhance the precision of hot page identification, thereby facilitating efficient data migration. In contrast to these approaches, APEX merely leverages PEBS to drive proactive remote memory prefetching, shifting the utility of hardware-assisted profiling from reactive management to predictive acceleration.

VI. Discussion

Single-tenant scope and future multi-tenant directions. Our current implementation primarily targets a single-tenant setting. Since correlation rules are learned within one address space, they are not readily transferable across tenants. In multi-tenant environments, prefetching competes for scarce local DRAM and network bandwidth, potentially causing cross-tenant interference, congestion, and fairness/QoS issues. As future work, we plan to extend APEX to multi-tenant prefetching by introducing tenant-aware isolation and quotas (for rule tables and swapcache budget), global bandwidth scheduling and rate limiting, and admission/adaptation mechanisms driven by per-tenant precision and benefit signals, thereby improving aggregate throughput while preserving isolation and fairness.

Limitations of PEBS sampling. APEX relies on PEBS to sample VAs online and to learn page-granular access correlations. This design enables deployment on commodity hardware with manageable overhead, but it also introduces limitations. VAs are stable only within a given mapping interval; dynamic memory operations, allocator churn, or rapid phase transitions may invalidate previously learned correlations. Moreover, VAs do not directly encode system state such as a page's current residency or NUMA placement. As a result, APEX must reconcile sampled VAs with kernel paging state at fault time. Sampling sparsity and phase changes can further reduce rule effectiveness, increasing reliance on fallback readahead and thereby limiting the achievable performance gains.

VII. Conclusion

This paper presents APEX, an adaptive prefetching strategy driven by application memory access patterns. By performing fine-grained tracking based on PEBS, APEX employs a CBF to intelligently identify cold pages. Building upon this identification, the system constructs corresponding access rules for these pages. APEX further introduces a dual-path decision framework to determine whether to invoke our pattern-based prefetching sequence or to fall back on the default Linux VMA-based policy. Evaluations are conducted on multi worklaods demonstrate that APEX enhances prefetching efficiency and overall system performance in the context of far memory systems.

In the future, we plan to design an more accurately and adaptive cold page detection method in the CBF frequency tracker module and avoid building the tracker for hot page to some extent.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Theodore Wong, for their constructive comments. This work was partially supported by the National Natural Science Foundation of China under grants 62502391. Dr Xiao Zhang and Shujie Han are the corresponding authors.

References

[1] J. Mei, S. Sun, C. Li, C. Xu, C. Chen, Y. Liu, J. Wang, C. Zhao, X. Hou, M. Guo *et al.*, "Flowwalker: a memory-efficient and high-performance gpu-based dynamic graph random walk framework," *arXiv preprint arXiv:2404.08364*, 2024.

[2] Y. Pu, C. Wang, X. Hou, C. Xu, J. Liu, J. Wang, M. Guo, and C. Li, "M 2 sn: Adaptive and dynamic multi-modal shortcut network architecture for latency-aware applications," in *IEEE ICME*. IEEE, 2024, pp. 1–6.

[3] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proc. of the 18th ACM PPoPP*, 2013, pp. 135–146.

[4] P. Wang, C. Li, J. Wang, T. Wang, L. Zhang, J. Leng, Q. Chen, and M. Guo, "Skywalker: Efficient alias-method-based graph sampling and random walk on gpus," in *30th PACT*. IEEE, 2021, pp. 304–317.

[5] J. Wang, H. Gu, J. Yu, Y. Song, X. He, and Y. Song, "Research on virtual machine consolidation strategy based on combined prediction and energy-aware in cloud computing platform," *Journal of Cloud Computing*, 2022.

[6] J. Wang, J. Yu, Y. Song, X. He, and Y. Song, "An efficient energy-aware and service quality improvement strategy applied in cloud computing," *Cluster Computing*, 2023.

[7] Amazon, "Iaas," 2024, https://aws.amazon.com/cn/what-is/iaas.

[8] "Huawei cloud. huawei cloud object storage service (obs)," 2019, https://www.huaweicloud.com/product/obs.html.

[9] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan *et al.*, "When cloud storage meets {RDMA}," in *USENIX NSDI*, 2021, pp. 519–533.

[10] N. Bleier, M. H. Mubarik, G. R. Swenson, and R. Kumar, "Space microdatacenters," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 900–915.

[11] D. Xu, M. Xu, C. Lou, L. Zhang, G. Huang, X. Jin, and X. Liu, "Socflow: Efficient and scalable dnn training on soc-clustered edge servers," in *Proc. of the 29th ACM ASPLOS*, 2024, pp. 368–385.

[12] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–34, 2022.

[13] L. Zhang, W. Feng, C. Li, X. Hou, P. Wang, J. Wang, and M. Guo, "Tapping into nfv environment for opportunistic serverless edge function deployment," *IEEE Transactions on Computers*, vol. 71, no. 10, pp. 2698–2704, 2021.

[14] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?" in *Proc. of the 15th Eurosys*, 2020, pp. 1–16.

[15] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proc. of the 26th ACM ASPLOS*, 2021, pp. 79–92.

[16] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *USENIX OSDI*, 2016, pp. 249–264.

[17] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *USENIX NDSI*, 2017, pp. 649–667.

[18] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid *et al.*, "Software-defined far memory in warehouse-scale computers," in *Proc. of the 24th ASPLOS*, 2019, pp. 317–330.

[19] H. Al Maruf and M. Chowdhury, "Effectively prefetching remote memory with leap," in *USENIX ATC*, 2020, pp. 843–857.

[20] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "{LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation," in *USENIX OSDI*, 2018, pp. 69–87.

[21] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu, "Semeru: a {Memory-Disaggregated} managed runtime," in *USENIX OSDI*, 2020, pp. 261–280.

[22] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "{AIFM}:{High-performance},{application-integrated} far memory," in *USENIX OSDI*, 2020, pp. 315–332.

[23] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "{FaRM}: Fast remote memory," in *USENIX NSDI*, 2014, pp. 401–414.

[24] Q. Li, H. Huang, Y. Liu, Y. Xia, J. Zhang, M. Zhou, X. Feng, H. Cui, Q. Chen, Y. Shan *et al.*, "Beehive: A scalable disaggregated memory runtime exploiting asynchrony of multithreaded programs," in *USENIX NSDI*, 2025, pp. 167–187.

[25] C. Wang, H. Ma, S. Liu, Y. Qiao, J. Eyolfson, C. Navasca, S. Lu, and G. H. Xu, "{MemLiner}: Lining up tracing and application for a {Far-Memory-Friendly} runtime," in *USENIX OSDI*, 2022, pp. 35–53.

[26] Https://www.computeexpresslink.org/.

[27] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal *et al.*, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proc. of the 28th ACM ASPLOS*, 2023, pp. 574–587.

[28] D. Sorin, M. Hill, and D. Wood, *A primer on memory consistency and cache coherence*. Morgan & Claypool Publishers, 2011.

[29] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *Proc. of the 11th ISCA*, 1984, pp. 348–354.

[30] H. Li, K. Liu, T. Liang, Z. Li, T. Lu, H. Yuan, Y. Xia, Y. Bao, M. Chen, and Y. Shan, "Hopp: Hardware-software co-designed page prefetching for disaggregated memory," in *IEEE HPCA*. IEEE, 2023, pp. 1168–1181.

[31] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Communications of the ACM*, vol. 60, no. 4, pp. 48–54, 2017.

[32] Y. Kim, H. Kim, and W. J. Song, "Nomad: Enabling non-blocking os-managed dram cache via tag-data decoupling," in *IEEE HPCA*. IEEE, 2023, pp. 193–205.

[33] L. Chen, S. Liu, C. Wang, H. Ma, Y. Qiao, Z. Wang, C. Wu, Y. Lu, X. Feng, H. Cui *et al.*, "A tale of two paths: Toward a hybrid data plane for efficient {Far-Memory} applications," in *USENIX OSDI*, 2024, pp. 77–95.

[34] Y. Qiao, C. Wang, Z. Ruan, A. Belay, Q. Lu, Y. Zhang, M. Kim, and G. H. Xu, "Hermit:{Low-Latency},{High-Throughput}, and transparent remote memory via {Feedback-Directed} asynchrony," in *USENIX NSDI*, 2023, pp. 181–198.

[35] W. Yoon, J. Ok, J. Oh, S. Moon, and Y. Kwon, "Dilos: Do not trade compatibility for performance in memory disaggregation," in *Proc. of the Eighteenth Eurosys*, 2023, pp. 266–282.

[36] Y. Pan, Y. Lala, M. Unal, Y. Ren, S.-s. Lee, A. Bhattacharjee, A. Khandelwal, and S. Kashyap, "Scalable far memory: Balancing faults and evictions," in *Proc. of the 31st ACM SOSP*, 2025, pp. 136–152.

[37] S. Liang, R. Noronha, and D. K. Panda, "Swapping to remote memory over infiniband: An approach using a high performance network block device," in *2005 IEEE Cluster Computing*. IEEE, 2005, pp. 1–10.

[38] I. Cutress and A. Frumusanu, "Amd 3rd gen epyc milan review: A peak vs per core performance balance," *Retrieved July*, vol. 30, p. 2024, 2021.

[39] C. Branner-Augmon, N. Galstyan, S. Kumar, E. Amaro, A. Ousterhout, A. Panda, S. Ratnasamy, and S. Shenker, "3po: Programmed far-memory prefetching for oblivious applications," *arXiv preprint arXiv:2207.07688*, 2022.

[40] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[41] T. Lee, S. K. Monga, C. Min, and Y. I. Eom, "Memtis: Efficient memory tiering with dynamic page classification and page size determination," in *Proc. of the 29th ACM SOSP*, 2023, pp. 17–34.

[42] J. Choi, S. Blagodurov, and H.-W. Tseng, "Dancing in the dark: Profiling for tiered memory," in *2021 IEEE IPDPS*. IEEE, 2021, pp. 13–22.

[43] P. Duraisamy, W. Xu, S. Hare, R. Rajwar, D. Culler, Z. Xu, J. Fan, C. Kennelly, B. McCloskey, D. Mijailovic *et al.*, "Towards an adaptable systems architecture for memory tiering at warehouse-scale," in *Proc. of the 28th ACM ASPLOS*, 2023, pp. 727–741.

[44] D.-J. Oh, Y. Moon, E. Lee, T. J. Ham, Y. Park, J. W. Lee, and J. H. Ahn, "Maphea: A lightweight memory hierarchy-aware profile-guided heap allocation framework," in *Proc. of the 22nd ACM ASPLOS*, 2021, pp. 24–36.

[45] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter, "Hemem: Scalable tiered memory management for big data applications and real nvm," in *Proc. of the ACM SIGOPS 28th SOSP*, 2021, pp. 392–407.

[46] mm, "Swap: Vma based swap readahead," https://lwn.net/Articles/716296/.

[47] Y. Huang, Z. Guo, and Y. Zhang, "Deep-learning-driven prefetching for far memory," *arXiv e-prints*, pp. arXiv–2506, 2025.

[48] D. Xu, J. Ryu, K. Shin, P. Su, and D. Li, "{FlexMem}: Adaptive page profiling and migration for tiered memory," in *USENIX ATC*, 2024, pp. 817–833.

[49] K. Song, J. Yang, Z. Wang, J. Zhao, S. Liu, and G. Pekhimenko, "Hybridtier: an adaptive and lightweight cxl-memory tiering system," ser. ASPLOS '25, 2025.
[50] T. Hurst., "Bloom filter calculator." Accessed: 2025, https://hur.st/bloomfilter/.
[51] Y. Sun, J. Kim, Z. Yu, J. Zhang, S. Chai, M. J. Kim, H. Nam, J. Park, E. Na, Y. Yuan *et al.*, "M5: Mastering page migration and memory management for cxl-based tiered memory systems," in *Proc. of the 30th ACM ASPLOS*, 2025, pp. 604–621.
[52] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *International conference on database theory*. Springer, 2005, pp. 398–412.
[53] "Stream: Sustainable memory bandwidth in high performance computers," https://www.cs.virginia.edu/stream/.
[54] T. Oliphant., "Numpy: A guide to numpy." USA: Trelgol Publishing, 2006.
[55] "Scikit-learn library," https://scikit-learn.org/stable/index.html.
[56] "C++ standard library headers," https://cppreference.com/.
[57] J. Löff, D. Griebler, G. Mencagli, G. Araujo, M. Torquati, M. Danelutto, and L. G. Fernandes, "The nas parallel benchmarks for evaluating c++ parallel programming frameworks on shared-memory architectures," *Future Generation Computer Systems*, vol. 125, pp. 743–757, 2021.
[58] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "Diskseen: Exploiting disk layout and access history to enhance i/o prefetch." in *USENIX ATC*, vol. 7, 2007.
[59] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun, "I/o acceleration with pattern detection," in *Proc. of the 22nd HPDC*, 2013, pp. 25–36.
[60] G. Soundararajan, M. Mihailescu, and C. Amza, "{Context-Aware} prefetching at the storage server," in *USENIX ATC*, 2008.
[61] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proc. of the 26th ACM ASPLOS*, 2021, pp. 559–572.
[62] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr, "Fast restore of checkpointed memory using working set estimation," in *Proc. of the 7th ACM SIGPLAN/SIGOPS VEE*, 2011, pp. 87–98.
[63] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in *International Conference on Machine Learning*. PMLR, 2018, pp. 1919–1928.
[64] A. Laga, J. Boukhobza, M. Koskas, and F. Singhoff, "Lynx: A learning linux prefetching mechanism for ssd performance model," in *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2016, pp. 1–6.
[65] Z. Song, D. S. Berger, K. Li, and W. Lloyd, "Learning relaxed belady for content distribution network caching," in *USENIX NSDI*, 2020, pp. 529–544.
[66] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *Proc. of the 15th ACM SOSP*, 1995, pp. 79–95.
[67] A. Tomkins, R. H. Patterson, and G. Gibson, "Informed multi-process prefetching and caching," *ACM SIGMETRICS Performance Evaluation Review*, vol. 25, no. 1, pp. 100–114, 1997.
[68] S. VanDeBogart, C. Frost, and E. Kohler, "Reducing seek overhead with application-directed prefetching." in *USENIX ATC*, 2009.
[69] K. Harty and D. R. Cheriton, "Application-controlled physical memory using external page-cache management," in *Proc. of the 15th ACM ASPLOS*, 1992, pp. 187–197.
[70] H. Ma, S. Liu, C. Wang, Y. Qiao, M. D. Bond, S. M. Blackburn, M. Kim, and G. H. Xu, "Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters," in *Proc. of the 43rd ACM PLDI*, 2022, pp. 92–107.