MARB: Bridge the Semantic Gap between Operating System and Application Memory Access Behavior

Haifeng Li*†§, Ke Liu*§, Ting Liang*†, Zuojun Li*†, Tianyue Lu*, Yisong Chang*,
Hui Yuan¶, Yinben Xia¶, Yungang Bao*†, Mingyu Chen*†‖, Yizhou Shan‡
*State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences
‖Zhongguancun Laboratory,Beijing,China †University of Chinese Academy of Sciences ¶Huawei Technologies ‡Huawei Cloud

***Abstract*—The virtual memory subsystem (VMS) is a long-standing and integral part of an operating system (OS). It plays a vital role in enabling remote memory systems over fast data center networks and is promising in terms of transparency and generality. Specifically, these systems use three VMS mechanisms: demand paging, page swapping, and page prefetching. However, the VMS inherent data path is costly, which takes a huge toll on performance. Despite prior efforts to propose page swapping and prefetching algorithms to minimize the occurrences of the data path, they still fall short due to the semantic gap between the OS and applications – the VMS has limited knowledge of its running applications' memory access behaviors.**

In this paper, orthogonal to prior efforts, we take a fundamentally different approach by building an efficient framework to collect full memory access traces at the local bus, and make them available to the OS through CPU cache. Consequently, the page swapping and page prefetching can use this trace to make better decisions, thereby improving the overall performance of systems. We implement a proof-of-concept prototype on commodity x86 servers using a hardware-based memory tracking tool. To showcase our framework's benefits, we integrate it with a state-of-the-art remote memory system and the default kernel page eviction subsystem. Our evaluation shows promising improvements.

I. Introduction

Nowadays, the virtual memory subsystem (VMS) plays a vital role in enabling remote memory systems over modern networks with microsecond-scale latencies. This is because it is a promising and practical approach due to its *generality* and *transparency*. For instance, cloud vendors leverage the VMS to transparently swap out cold memory to remote in order to save cost [1], [2], kernel-based disaggregated memory systems tightly couple with the VMS for remote access [3]–[5]. Moreover, heterogeneous and multi-tiered memory systems, built on top of emerging memories such as PM [7] or interconnects such as CXL [8], leverage the VMS for data migration or swapping.

However, the existing VMS data path is costly, has limited parallelism due to coarse-grained locking and synchronous function calls. Despite the efforts to optimize the data path, this drawback is not fundamental and cannot be completely avoided [6]. Prior works thus proposed better page prefetching and eviction algorithms to avoid the data path as much as possible [3], [4]. However, they still fall short and their advantages can be offset by the second limitation – the page swapping and prefetching are not able to accurately identify what pages to evict or prefetch. This is because they train algorithms with limited memory access trace from page faults. One possible workaround is using a daemon to periodically scan the access bit in page table to approximate an LRU history [1]. But this approach not only incurs non-trivial overheads due to TLB shootdown and pinned thread but also produces coarse-grained and stale trace, let alone sufficient memory trace in real-time.

We observe that this limitation stems from the semantic gap between OS and hardware: the CPU hardware only exposes coarse-grained and stale access information via page faults and page tables, hence the OS has limited knowledge of its running applications' memory access history and has to use costly software approaches to approximate application access pattern. Our observation is driven by a simple insight that *rich access history originated from higher tiers in the memory hierarchy enables the lower tiers to better model the memory access pattern, thereby making better eviction or prefetching predictions.* For instance, if OS knows all the last-level cache (LLC) misses, the OS can build a precise eviction and prefetching algorithm.

We therefore propose a fundamentally different approach by collecting full memory access traces (*i.e.*, LLC misses) at the local bus and feeding them to the OS continuously and efficiently. Consequently, the VMS has abundant supply of real-time memory access information to improve its algorithms on page swapping and prefetching. In specific, we build a software-hardware co-designed framework called Memory Access Record Buffering (MARB). By design, MARB deploys a *record unit* in the memory controller (MC). The record unit can record all memory accesses and generate compact access records. To avoid making another round-trip to the main memory and to reduce memory bandwidth usage, the record unit directly sends the generated records into the CPU cache [11]. To avoid polluting the CPU cache, we reserve a dedicated cache partition to store the trace using way-partitioning [12] and page coloring [13]. The trace is organized as a circular buffer, with the MC writing to the tail and a dedicated OS thread reading from the head whenever the buffer is not empty. Since the trace is generated in massive volume, we employ a *filter table* to screen noise and aggregate accesses to the same pages so as to extract hot pages. These pages are then pipelined into a *reverse page table*, which translates physical addresses back to virtual addresses (VA) and associated process identifications (PID).

We build two use cases to demonstrate MARB's benefits. We first improve a state-of-the-art remote memory system [3] by replacing its default sequential prefetching policy with a stream-based prefetching algorithm powered by MARB. Similarly, we

§Equal contribution

978-3-9819263-7-8/DATE23/© 2023 EDAA

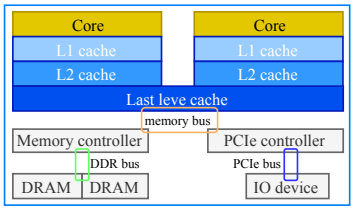

Figure 1. A typical server layout with CPUs, LLC, MC, DRAM, PCIe devices connected via various buses.

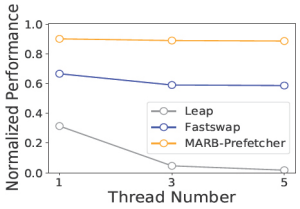

Figure 2. Completion time performance normalized to local scenario with no remote memory.

improve the default Linux kernel LRU lists by replacing its access bit-based approach with the rich access trace from MARB. There are alternative ways to use the trace to highly optimized for use cases. *e.g.*, users could employ a different or even opt-out the filter table to parse the raw trace instead.

Since the MC is vendor-locked, we build a proof-of-concept prototype on commodity x86 servers for design validation and evaluation (§V). To emulate the record unit's role, we leveraged a hardware-based memory tracking tool, HMTT [15] and Data Direct I/O technology (DDIO) [11]. We build the MARB's software in user space. We believe MARB's hardware design is lightweight and generic hence could be easily integrated into a CPU chip or an MC of various CPU platforms.

We evaluate MARB and the two use cases using a set of microbenchmarks and many large-scale in-memory applications. Overall, MARB helps improve VMS-based systems. For instance, when half of an application's working set is disaggregated, MARB improves a state-of-the-art remote memory system by 59% with over 90% prefetching accuracy and coverage. Also, MARB-enhanced page eviction eliminates swapping events and delivers over 30% improvements over the default kernel.

In summary, we make the following contributions:

❶ We propose a lightweight hardware unit to collect access trace in the memory controller and save trace directly to the cache for software to improve VMS performance.

❷ We build a proof-of-concept prototype using a set of novel techniques and a hardware tracking tool to emulate and validate MARB, and demonstrate that it can efficiently capture and expose trace to upper layer systems.

❸ We use MARB to improve a disaggregated memory system and the default kernel eviction subsystem, which confirms that MARB dramatically improves VMS-based systems.

II. Background

We first walk through the modern server layout, laying out a solid background of MARB. We then discuss the two use cases that can benefit from using MARB.

A. Modern Server Architecture

Figure 1 shows a typical server layout. The LLC, MC, and PCIe controller are interconnected through an internal local bus. The MC talks to DRAM chips via a standard DDR bus. The PCIe controller communicates with I/O devices via a PCIe bus. Clearly, the MC knows the full memory accesses from all CPUs. As writing trace to DRAM directly consumes extra memory bandwidth, and takes another round-trip from DRAM to the LLC when the OS reads the trace, for MARB, we let the MC write the trace into the LLC directly over the local bus.

Table I
ACCURACY AND COVERAGE WITH DIFFERENT NUMBER OF THREADS.

Name	Acc(P1)	Cov(P1)	Acc(P3)	Cov(P3)	Acc(P5)	Cov(P5)
Leap	0.99	0.87	0.89	0.85	0.86	0.84
Fastswap	0.99	0.87	0.87	0.82	0.85	0.81
MARB-Prefetcher	0.99	0.99	0.99	0.99	0.99	0.99

Direct Cache Access. Typically, I/O devices exchange data with CPU by using Direct Memory Access (DMA) to transfer data into the DRAM. The CPU then reads the data over the local bus and the DDR bus. However, this workflow involves many unnecessary data movements. The Direct Cache Access technique is therefore proposed to allow the PCIe controller to access the LLC directly, such as DDIO [16] in Intel-based platform. Other CPU architectures have similar ones [17].

Cache Partition. Cache partition techniques mitigate cache interference and ensure fairness among co-running workloads There are three techniques: 1) software-based, which uses *page coloring* to partition the cache among sets [13]; 2) hardware-based, such as hardware way-partitioning (*e.g.*, Cache allocation technology (CAT) [12]); 3) hybrid one which combines set and way partitioning [18] , divvying up the LLC space. For MARB, we use a hybrid one, *i.e.*, page coloring and CAT, along with DDIO to reserve cache size to mitigate cache pollution.

B. Disaggregated Memory Systems

The VMS is selected as a key enabler for remote memory systems for cloud vendors due to its generality and transparency [1], [2]. However, the slow VMS data path inevitably hampers applications from performing efficiently. As a case study, we evaluate Fastswap [3], and find that Fastswap takes roughly 9μs to read a page from remote using a single core. The latency increases significantly when increasing the number of threads (*e.g.*, 16μs for 8 threads). It is clear that Fastswap fails to scale. This is due to the coarse-grained locking and synchronous data path designs inherent in existing VMS.

One way to overcome the above VMS overheads is to use *page prefetching* by reading pages from remote beforehand so as to avoid invoking the VMS data path as much as possible. Fastswap uses the default kernel read ahead prefetching policy to read the subsequent seven pages following a faulting page [3]. Presumably, if we run a program that scans a memory region sequentially, the above simple policy *should* have perfect accuracy and coverage (definitions in §VI). However, as Table I shows, both metrics decrease as we increase parallelism, *e.g.*, the accuracy and coverage drop to 85% and 81%, respectively with 5 threads. This causes a worse completion time than a prefetcher with full memory access knowledge from MARB (MARB-Prefetcher in §IV-B) in Figure 2. The same observation is found in Leap [4], an optimized disaggregated memory system with an online, majority-based prefetching algorithm.

The root cause is that both Fastswap and Leap only train their prefetch algorithms with infrequent and stale access information from page faults, thus cannot accurately differentiate the accesses from different threads. To solve it we integrate MARB with Fastswap by replacing its default sequential prefetching policy with a stream-based one trained using the rich application access trace provided by MARB. Second, MARB enables any prefetch algorithm to run in a separate asynchronous data path, independent from VMS one, to achieve scalability.

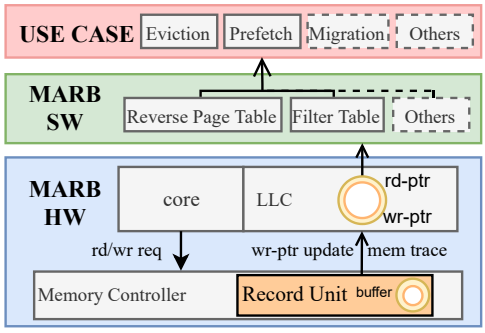

Figure 3. MARB framework architecture.

C. Page Swapping and Eviction

Page swapping plays a key role in modern data centers [1], [2]. *e.g.*, both Google and Meta rely on it to identify cold memory. Generally, the page swapping uses approximate LRU lists to model memory access pattern and select which pages to evict. However, the lists are ordered by scanning the access bit in page tables periodically, which produces unreliable, infrequent, and coarse-grained page-level accesses. Consequently, LRU lists fail to capture the actual memory access pattern. We improve the default LRU lists by replacing the access bit-based approach with real-time rich memory trace from MARB. Moreover, when there is an eviction request, we prioritize pages with streaming access pattern as they are not likely to be used again compared to randomly accessed pages.

III. Design

MARB is a software-hardware co-designed framework. Figure 3 depicts the overall architecture. We deploy a hardware-based record unit in the MC. This unit captures all memory accesses originated from LLC misses. We exploit direct cache access and cache partition techniques, the record unit sends the trace into a reserved cache partition through the local bus directly. We organize the trace as a circular buffer, with the MC writing to the tail (*write_ptr*) while a dedicated MARB thread continuously reading from the head (*read_ptr*). Once traces are obtained, the MARB thread aligns the physical addresses to page-granularity and uses a filter table to screen noise and aggregate accesses to the same pages so as to extract a set of hot pages. Then it sends them into a reverse page table, which translates physical addresses back to VA and associated PID.

A. Record Unit

The record unit plays a key role in MARB with two tasks: 1) It has a trace tracking unit snooping the memory request from LLC and generates raw memory trace. 2) It has a parse-and-forward unit that packages the trace into compactly formatted records, which are then forwarded to the CPU cache for online analysis. The record unit is the only new hardware logic added by MARB. By design, the record unit is integrated with the MC. For prototyping, we use a memory tracking tool for task 1 and implement task 2 in a separate FPGA board (§V).

As Figure 3 shows, when the MC receives an LLC miss, the record unit captures this request and generates a 6 B record. Since the local bus's transfer granularity is 64 B, the record unit can pack at most ten complete records and one partial record at a time. The packed records are then forwarded into the LLC directly using one local bus packet (upper half in Figure 4). To

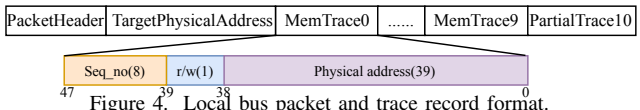

Figure 4. Local bus packet and trace record format.

prevent the OS from reading partial records from the circular buffer, the record unit advances the write pointer in a batched fashion, *i.e.*, once per three bus packets or 32 complete records. Thus, it only adds 200 B overhead for records batching. To support above operations, the record unit only requires three registers, namely the start physical address (MARB cache is indexed by physical address), MARB cache size, and the batch size. MARB is configured to be accessed by specific processes. Protect MARB is equivalent to protect inter-process address.

Cache size. The records are sent and stored to a cache whose size can be set to a minimum, $2 \times batch\_size$, if the MARB thread processes trace fast enough. By prototyping, the records are sent by PCIe, which requires a larger $batch\_size$ (1 KB) to ensure high throughput, thus a small cache of 2 KB is sufficient to avoid buffer overflow for all the tested applications (§V).

Cache partition. To avoid cache pollution, MARB cache is reserved by hybrid cache partitioning. We first use way-partition like Intel CAT [12] to reserve 10% LLC (3.5 MB in our platform). Since MARB only needs 2KB, we use page coloring to return the rest back to applications (*i.e.*, $3.5MB - 2KB$).

Figure 4 shows the local bus packet format and the structure of a record. The bus packet contains some packet headers, a physical address, together with several records. This format is bus-specific. Each record is 6 bytes, with a sequence number (8 bits), a record type indicating whether it is a read or write request (1 bit), and a cache line-aligned physical address (39 bits). The sequence number is increasing monotonically. The MARB software uses it to determine whether any trace is lost or overwritten, and takes extra measures to handle the sequence number overflow and loopback during runtime. Essentially, we trade some software complexity for hardware efficiency.

B. Filter Table

MARB software continuously monitors and reads the circular buffer for incoming traces from the record unit, which performs several tasks. First, it maps cache line-aligned address into page-aligned. Second, it will drop all records whose type is write, because every write LLC miss results in a read to memory and a write to cache. Thus, it is sufficient to count read requests. Once access frequency of a particular page exceeds a configurable threshold, we extract it as a hot page. We employ a busy flag to avoid repetitive extractions within a short period. Moreover, users of MARB could design their own filter table [19], or bypass the default filter table partially or entirely (the dashed box in Figure 3). We leave these customizations for future work. As measured, filter table can process the trace generation rate over 10GBps (see §VI-C), which is sufficient for most use cases. Additionally, the use cases using page-level trace also can tolerate trace loss for streams detection (§IV).

C. Reverse Page Table

The memory address emitted by the filter table is still physical address (PA). However, most subsystems cannot use PA at all and have to deal with virtual address (VA). Reverse mapping subsystem (rmap) in Linux kernel is designed for this

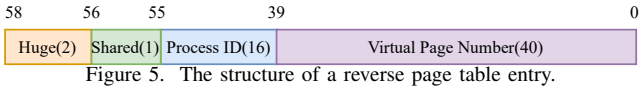

Figure 5. The structure of a reverse page table entry.

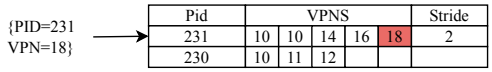

Figure 6. The structure of a stream table (ST), where *start_page* and *end_page* denote the first and end page in VPN array. A hot page hits ST first entry. The dominant stride is 2, thus the prefetching candidate is $18 + 100 \times 2$.

purpose, but the rmap takes at least four memory accesses to complete a single translation (*i.e.*, it will read `struct page`, `struct vma`, and a few other auxiliary variables), which is unnecessarily complicated and inefficient for our purpose.

In response, we build reverse page table (RPT), a lightweight mechanism tailored for `MARB`'s requirements. Similar to prior works [15], the RPT is organized as a flat array indexed by physical page numbers. Figure 5 shows the structure of an RPT entry, which has a huge page flag (2 bits) indicates if the page size is 4KB, 2MB, or 1GB, a shared page flag (1 bit) indicates whether the physical page is shared by multiple processes, followed by a PID (16 bits) and a virtual page number (40 bits). We perform address translation at 4KB granularity. The huge page flag indicates whether a regular page is part of a huge page. For most cases, one RPT translation requires only one memory access, except for shared pages, where we would consult the kernel rmap for details. Fortunately, page sharing is uncommon in remote memory settings.

IV. Use Cases

Based on the full memory trace from `MARB` we build two use cases to demonstrate how `MARB` can help VMS-based systems, *i.e.*, a new LRU-based eviction subsystem and a prefetching engine for an RDMA-based disaggregated memory system [3].

A. Page Eviction

The off-the-shelf LRU page lists in the Linux kernel are ordered using infrequent and stale access bits (§II). We build a new subsystem called `MARB-Evictor` to improve it. First, to avoid kernel complexity we build the eviction data path in the user space. Once the evictor detects that the whole server is under memory pressure, it will select a set of pages to evict and instruct the kernel to finalize the eviction process. Second, the evictor uses an LRU-based page list ordered based on the full memory trace from `MARB`, which closely resembles application access behavior. Finally, the evictor employs an optimization to differentiate various access patterns. While selecting pages to evict, we prioritize pages that follow a streaming access pattern. This is based on a simple insight that pages already used in a streaming access pattern are less likely to be used again compared to pages following a random access pattern. We use the page stream table to identify pages streams (§IV-B).

B. Disaggregated Memory Prefetcher

Based on the full trace from `MARB`, we build a stream-based prefetcher in user space called `MARB-Prefetcher`. `MARB` enables the prefetcher to instruct Fastswap to prefetch from remote asynchronously (without waiting for page faults). We also integrate the above `MARB-Evictor` into Fastswap.

A core concept in our prefetcher is page stream, which describes a sequence of accesses following a fixed stride. An application can have multiple page streams simultaneously.

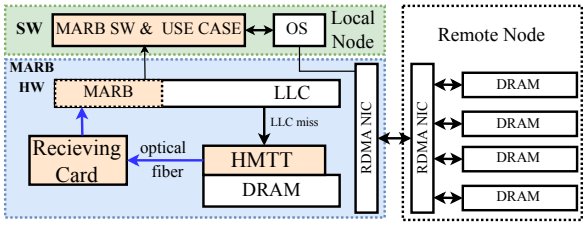

Figure 7. `MARB` Proof-of-concept Prototype.

Our prefetcher maintains a stream table (ST) with each entry representing a page stream. As shown in Figure 6, the ST has 64 entries which is sufficient in most scenarios. The entries are managed using an LRU policy. Furthermore, an ST entry can record up to five virtual page numbers (VPNs).

Our prefetcher works as follows: when {VPN,PID} arrives, we first check whether the distance between a hot page's VPN and a particular stream's $start\_page$ is less than a predefined value Δ_{stream}, the prefetcher would determine that this hot page belongs to that stream. Consequently, the prefetcher will enqueue this page's VPN into the matched stream's VPN array stored in its ST entry. When the array is full, we will calculate the distribution of stride values (*i.e.*, the distance between VPNs within the array) and find a dominant stride. With that, the prefetcher determines that VPNs for prefetching are stream's $end\_page + i \times stride$, where i is prefetch offset that indicates how far in time should we prefetch. To incorporate host and network delays, we set $i = 100$ to ensure the page arrives in time before the application accesses it. We leave the online adjustment of i to the future if those delays are fluctuating.

V. Implementation

`MARB` hardware. It does not need to modify the core and cache. It only adds the recording logic of memory trace in MC and uses less than 200B for batching records (§III-A).

`MARB` software and user cases. Our principle is to implement software functionalities (RPT and filter table) and user cases in the user space. This allows us to iterate quickly and upgrade `MARB` seamlessly. We minimize kernel changes: 1) we scan existing page tables to initialize RPT, and add callbacks to VMS functions (*e.g.*, `pte_clear`, `set_pte`) to keep RPT updated; 2) we improve the kernel swap interface, as it is limited to synchronous data path and coarse-grained locking. First, it sends swap requests like prefetching or eviction requests to an RDMA backend. At this point, we return control back to the calling thread immediately without waiting for the request to finish. Once the request finishes, the RDMA NIC sends an interrupt and interrupt handler (built on top of the `workqueue` subsystem) will *finalize PTE related operations*. Second, we replace a coarse-grained PMD lock with a fine-grained per-page lock which greatly increases parallelism.

Hardware platform. Most MCs are vendor-locked and packaged with the CPU and LLC into a single die, hence we cannot implement the record unit in MC. We therefore built a proof-of-concept prototype on commodity x86 servers, as shown in Figure 7. First, to emulate the trace tracking in the record unit, we attach a hardware-based memory tracking tool called `HMTT` [15] to the link between the MC and DRAM chips. The `HMTT` captures the memory trace in real time by snooping

Table II
APPLICATION WORKLOADS.

Workloads	Footprint(GB)	Cores	Bandwidth(GB/s)
Spark-GraphX (BFS,CC,LP,PR)	40	8	3 - 6
Spark-Bayes	37	8	3 - 6
Spark-K-Means	40	8	2 - 4
OMP-K-Means	3.2	4	2 - 3
High Performance Linpack	1.2	2	3 - 6
NPB (CG,FT,LU,MG,IS)	1 – 7	2	3 - 6
QuickSort	4	1	1 - 2

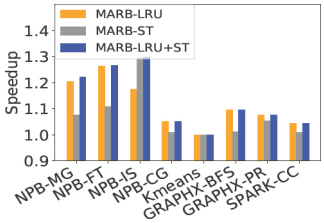

Figure 8. Completion time speedup.

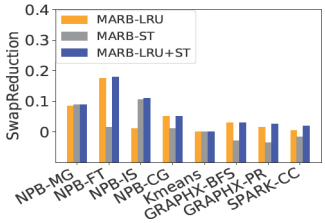

Figure 9. Swap reduction.

low-level DDR traffic at the DDRx memory bus. Second, to perform direct cache access required by the record unit, we use an FPGA-based PCIe receiving card that continuously receives records from HMTT through an optical fiber, and then sends them to LLC through DDIO. It takes 3-6μs to capture, parse, and forward the trace into the CPU cache. This delay only affects the timeliness of the trace but has no impact on application's memory access latency.

VI. EVALUATION

We first evaluate two MARB's use cases: page eviction and disaggregated memory prefetching with real applications. Then, we evaluate MARB's internal designs with microbenchmark.

Testbed and Workload. As Figure 7 shows, our testbed has two servers connected via RDMA, with one being local server running MARB and the other used as remote memory. Both servers have a 14-core Xeon E5-2680 CPU with 35 MB 20-way LLC, 64 GB DRAM, and a 56 Gbps Mellanox ConnectX-4 NIC. Table II shows the 14 large-scale in-memory applications used in our experiments. The footprint is their working set size.

A. Page Eviction

For the two use cases, we first evaluate MARB-Evictor, the MARB-enhanced page eviction (§IV-A). We use the Linux default eviction mechanism and its kswapd as the baseline. We evaluate MARB-Evictor with 1) the LRU-list backed by MARB's full trace (MARB-LRU), 2) the streaming access pattern (MARB-ST), and 3) both optimizations (MARB-LRU+ST). We use Fastswap [3] as the swap backend thus pages are evicted to remote memory. Compared to the baseline we report application completion time speedup and swap reduction defined as the reduced number of total swap events, higher the better.

Figure 8 and Figure 9 report the performance and swap reductions of 8 real applications with MARB-Evictor. Random pattern dominates the NPB-MG, NPB-FT and Spark-Graphx's access patterns, thus MARB-Evictor with MARB-LRU optimization performs better than applying MARB-ST only for them. In contrast, NPB-IS's access pattern is dominated by Stream pattern, thus MARB-Evictor with MARB-ST optimization achieves the best speedup (1.3x). Similarly, the reason behind the application speedup is because MARB-Evictor reduces the number of swap events for non-stream pages (Figure 9). As Kmeans's access is similar to Stream-only, none of MARB-Evictor optimizations works for Kmeans.

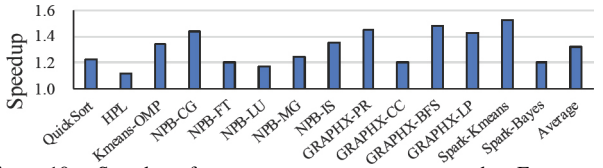

Figure 10. Speedup of MARB-Prefetcher compared to Fastswap.

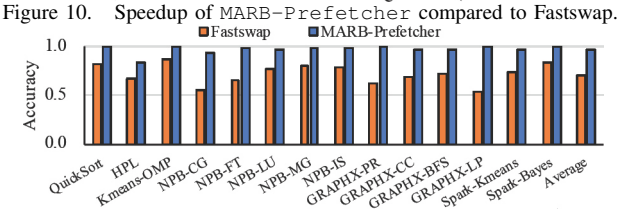

Figure 11. The prefetching accuracy of MARB-Prefetcher and Fastswap.

B. Disaggregated Memory Prefetcher

We integrate the MARB-enhanced prefetcher with Fastswap (MARB-Prefetcher) and compare it to original Fastswap and Leap using various real applications (Table II). However, Leap has much worse performance than the other two due to its slow data path implementation (*e.g.*, Figure 2). We thus omit Leap's results for better illustration. We use two metrics to measure prefetch performance. **Accuracy:** The ratio of total page hits and the total prefetched pages. **Coverage:** The ratio of the total page hits from the prefetched pages and the total number of remote accesses. We also use **Application completion time speedup** over original Fastswap to show MARB-Prefetcher's efficiency. We set the local memory to 50% of the application footprint for all tests.

Figure 10 shows the application completion time speedup with MARB-Prefetcher. For QuickSort and Kmeans-OMP, MARB-Prefetcher incurs no performance slowdown compared to the *local scenario* where no memory is disaggregated, even half of their working set is disaggregated. This is because MARB-Prefetcher can accurately predict the access pattern and prefetch pages into local DRAM asynchronously, completely eliminating page faults. The average acceleration of all workloads with MARB-Prefetcher is 32.2%. MARB-Prefetcher accelerates Fastswap by 48.2% at most, and 11.4% at least.

To understand where the boost comes from, we further compare their accuracy and coverage. Figure 11 shows that the average accuracy of MARB-Prefetcher is over 97%, which means that only a few false prefetches are issued and local memory is not polluted. However, the average accuracy of Fastswap is only 71.1%. Figure 12 shows the prefetching coverage of MARB-Prefetcher is divided into two parts: *StreamHit* and *SwapCacheHit*. *SwapCacheHit* is the number of prefetched pages issued upon page faults by the underlying remote system. Recall that MARB-Prefetcher is a complement to the remote memory system running on the same server. *StreamHit* is the number of prefetched pages issued once a stream is formed, which would not cause page faults. MARB has the best prefetching coverage for QuickSort and Kmeans, with more than 99% coverage, thus no page fault observed. The application completion time is very close to the local scenario.

We evaluate the completion time speedup over Fastswap if combining MARB-Evictor and MARB-Prefetcher together (MARB-ALL). As Figure 13 shows, MARB-ALL outper-

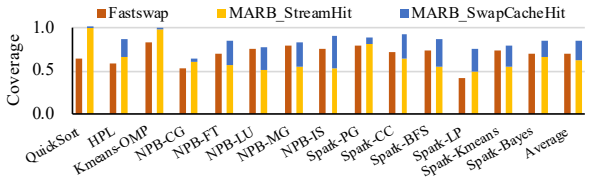

Figure 12. The prefetching coverage of `MARB-Prefetcher` and Fastswap.

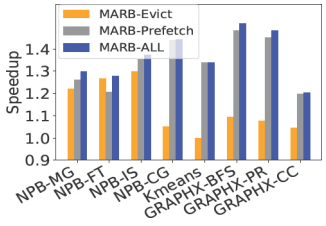

Figure 13. The completion time speedup if combining `MARB`'s prefetcher and evictor.

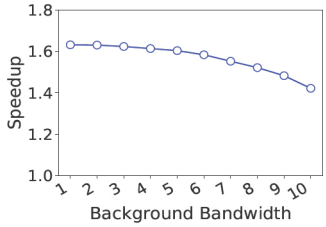

Figure 14. `MARB-Prefetcher`'s speedup over Fastswap with increasing memory bandwidth (GBps).

forms the other setups for all applications. except Kmeans. By sharing the same stream table, `MARB-Prefetcher` reduces swap events of `Stream` pages, `MARB-Evictor` reduces swap events of non-stream pages. As Kmeans has few non-streams, `MARB-ALL` performs similarly to `MARB-Prefetcher`.

Effect of memory bandwidth. To verify whether `MARB` is still effective with increasing memory bandwidth, we run a linear access microbenchmark with `MARB-Prefetcher` along with a background task generating memory traffic. The filter table would capture records generated by both testing programs but only analyze the ones from the microbenchmark. Figure 14 shows that, when the background bandwidth is less than 7 GBps, it has little effect on the `MARB-Prefetcher`.

C. *MARB Design Feasibility*

Record unit. Our record unit can generate at most 200 M records/sec. This is constrained by the FPGA frequency in the HMTT board [15]. Since each cache line is 64 B, the maximum memory bandwidth the record unit can track without loss is roughly 12.8 GBps. No record loss is found in our tests.

Filter table. The table is a flat array of one byte counters indexed by physical address used to aggregate memory accesses made to the same physical pages (§III-B). Our single-thread implementation' maximum process speed is 18.93 GBps, with a min of 10.81 GBps. The filter table performs the best when memory accesses are consolidated, resulting in cache hits, as it runs at L1 speed, but performs worst for random access.

VII. Related Work

Remote memory. Other than remote memory systems based on VMS interface [1]–[5]. There are systems built atop of interface such as objects [21], language runtime [20], and hardware [8]. However, they are not general or practical as VMS-based ones, either requiring nontrivial modification [8], [20], [21] or limiting to a few languages [20].

Prefetch algorithms. A large number of prefetching techniques have been proposed, like cache line granularity prefetcher [19] and can be realized as `MARB` use cases, as it provides sufficient memory access knowledge they need.

Intel PEBS. PEBS is a hardware feature in Intel CPUs. It provides accurate and fine-grained profiling. However, using PEBS has non-trivial CPU overhead. Whenever PEBS saves context information, the target workload would slow down by 200 – 300 ns. If using PEBS to sample every LLC miss, the application would take a huge performance toll. Moreover, PEBS can pollute cache and consume extra memory bandwidth [22].

VIII. Conclusion

This paper presents `MARB`, a software-hardware co-designed framework that can capture memory access trace and make them available to upper layer systems efficiently, bridging the semantic gap between OS and application memory access behavior. We use `MARB` to improve a disaggregated memory system and the kernel page eviction subsystem. We believe this work opens a door for improving various VMS-based systems.

Acknowledgement

This work was supported by Beijing Municipal Natural Science Foundation (No. 4212028), National Key Research and Development Plan of China (No. 2022YFB4500400), National Natural Science Foundation of China (No. 62090020 and 62072439), Strategic Priority Research Program of the Chinese Academy of Sciences (No. XDA0320300), and Shandong Provincial Natural Science Foundation (No. ZR2019LZH004). Corresponding authors: Ke Liu (liuke@ict.ac.cn) and Tianyue Lu (lutianyue@ict.ac.cn).

References

[1] Lagar-Cavilla, Andres, et al. "Software-defined far memory in warehouse-scale computers." ASPLOS 2019.
[2] Weiner, Johannes, et al. "TMO: transparent memory offloading in datacenters." ASPLOS. 2022.
[3] Amaro, Emmanuel, et al. "Can far memory improve job throughput?." EuroSys. 2020.
[4] Al Maruf, Hasan, and Mosharaf Chowdhury. "Effectively prefetching remote memory with leap." ATC. 2020
[5] Gu, Juncheng, et al. "Efficient memory disaggregation with infiniswap." NSDI. 2017
[6] Wang, Chenxi, et al. "Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory." NSDI. 2023
[7] Eisenman, Assaf, et al. "Reducing DRAM footprint with NVM in Facebook." EuroSys. 2018
[8] Li, Huaicheng, et al. "First-generation Memory Disaggregation for Cloud Platforms." arXiv preprint arXiv:2203.00241 (2022).
[9] Navin Shenoy. A milestone in moving data, 2019.
[10] Guo, Zhiyuan, et al. "Clio: A hardware-software co-designed disaggregated memory system." ASPLOS. 2022
[11] Intel.Intel®datadirectI/O(DDIO),2021.https://www.intel.com/content/www/us/en/io/data- direct- i- o- technology.html.
[12] Cache Allocation Technology, 2018. https://xenbits.xenproject.org/docs/unstable/features/intel psr cat cdp.html.
[13] Ye, Ying, et al. "Coloris: a dynamic cache partitioning system using page coloring." PACT. 2014
[14] Zhang, Xiao, Sandhya Dwarkadas, and Kai Shen. "Towards practical page coloring-based multicore cache management." EuroSys. 2009
[15] Huang, Yongbing, et al. "HMTT: A hybrid hardware/software tracing system for bridging the DRAM access trace's semantic gap." TACO. 2014
[16] Ibanez, et al. "The case for a network fast path to the CPU." Proceedings of the 18th ACM Workshop on Hot Topics in Networks. 2019.
[17] ARM. Arm DynamIQ shared unit technical reference manual. https://developer.arm.com/documentation/100453/0300/functional-description/l3-cache/cache-stashing, 2021.
[18] Wang, Xiaodong, et al. "SWAP: Effective fine-grain management of shared last-level caches with minimum hardware support." HPCA. 2017
[19] Pan, Haiyang, et al. "LSP: Collective Cross-Page Prefetching for NVM." DATE. 2021
[20] Wang, Chenxi, et al. "Semeru: A Memory-Disaggregated Managed Runtime." OSDI. 2020
[21] Ruan, Zhenyuan, et al. "AIFM:High-Performance,Application-Integrated Far Memory." OSDI. 2020
[22] Akiyama, Soramichi, and Takahiro Hirofuchi. "Quantitative evaluation of intel pebs overhead for online system-noise analysis." ROSS. 2017