HoPP: Hardware-Software Co-Designed Page Prefetching for Disaggregated Memory

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

***Abstract*—Memory disaggregation is a promising direction to mitigate memory contention in datacenters. To make memory disaggregation practical, prior efforts expose remote memory to applications transparently via virtual memory subsystem's swapping interface. However, due to the semantic gap between OS and applications – OS cannot know the memory accessing sequences of an application but via page faults. This approach has two limitations. First, it learns little from page faults' access history, which leads to sub-optimal prefetching predictions. Second, a page fault can still occur even if there is a prefetch-hit which leads to a large kernel overhead.**

To address such limitations, our key insight is to decouple the address capturing from page faults by collecting full memory access traces in the memory controller. Using this idea, we build HoPP – a hardware-software co-designed prefetching framework. HoPP adds hardware modules to the memory controller to feed sufficient hot pages to OS in real-time, which has three benefits in HoPP's software design: 1) it improves existing prefetching algorithms with simple revamps, also offers more insights to build better policies; 2) the prefetch algorithm can run as a separate data path alongside the normal remote data path via page faults, potentially hiding the swap latency from applications, and enabling fine-grained control over prefetching behaviors; 3) the prefetch-hit overhead can be eliminated by early page table entry (PTE) injection, *i.e.*, inject PTE for the prefetched page as soon as it returns. We implemented a proof-of-concept prototype using commodity servers along with a hardware-based memory tracking tool called HMTT to emulate a modified memory controller. Results show that compared to Fastswap and Leap, HoPP-optimized prefetching algorithm achieves over 90% accuracy and coverage, which leads to up to 59% completion time improvement for various datacenter applications.

I. Introduction

Datacenter in-memory applications such as big data analytics and caching have an increasing demand to access large amounts of memory [2], [18], [19], [62]. Their performances degrade when their working set fail to fit into local memory. Unfortunately, servers are facing memory capacity walls due to pin and power limitations [36], [50]. Meanwhile the average memory utilization in datacenters is low (*e.g.*, about 60% for Google and Alibaba clusters [37], [63]), abundant idle memory is beyond the reach of applications that desperately need it.

Memory disaggregation bridges this gap by organizing memory as an independent resource pool and making it available to applications. Disaggregation mitigates memory provisioning inefficiencies and improves resource utilization in datacenters [16], [22], [27], [34], [36], [55].

To make memory disaggregation practical, one important category of prior works relies on virtual memory subsystem (VMS) for *remote memory swapping* over high-speed network such as RDMA [7], [22], [34], [38], [49] – kernel-based remote system. Although this approach enables applications to *transparently*, without code changes, use remote memory, it relies on the page fault handler for remote accessing, which is synchronous and costly by adding swap overhead into the application's critical path. Despite recent efforts, the data path is still slow, *e.g.*, it takes $9\mu s$ for Fastswap [7], a recent kernel-based remote system, to read a remote page, which significantly degrades application performances.

Ideally, a remote system should minimize the remote memory access as much as possible so that the overhead from page faults is minimized. Therefore, along with the faulting page, the recent systems like Leap [38] and Fastswap [7] also *prefetch* pages into the *swapcache*. If they can be hit, less time is spent on the critical path. Unfortunately, their prefetchings still suffer a fundamental limitation: the semantic gap between OS and applications – *OS cannot know where (which memory address) an application is running but via page faults*. First, this makes prefetch algorithms trap into a paradox: naturally, we want fewer page faults. However, a prefetch algorithm then has limited memory access data to train on and eventually makes sub-optimal predictions (§II-B). On the other hand, obtaining sufficient knowledge of memory accesses to train prefetch algorithms means more page fault occurrences (*e.g.*, by setting more page fault points for profiling), making the application performance even worse. Second, even if their prefetched pages are hit eventually, we found that the prefetch-hit in swapcache is a synchronous process, resulting in a large kernel overhead (§II-C).

Recently, there are another three categories of remote memory systems proposed. However, 1) *application-integrated systems* sacrifices application transparency for better performance via explicit user control on data movement and prefetching [52]. 2) *Language-runtime managed systems* integrate remote systems into a user space language runtime or application kernel. But they are limited to a few languages [66]. 3) *Bus-*

§Equal contribution, Ke Liu (liuke@ict.ac.cn) is the corresponding author.

extended systems rely on emerging coherent interconnects like CXL-based platforms [1] for transparent remote accesses [11], [33]. However, extending bus interface is complex, requiring redesign existing coherence protocols [41], [47]. More importantly, they cannot use local DRAM for caching, which is critical for performance. Thus, for each hot data access, they have to access it remotely [11].

In this work, we strike a balance between these four categories with minimal hardware change, *i.e.*, not only retaining application transparency and high-speed local DRAM cache, but also achieving efficient prefetching. The key insight is that we can *decouple the memory access address generation from the page fault, by collecting real time memory access trace in memory controller (MC) and make it available to OS*. As a result, a prefetch algorithm has an abundant supply of real-time addresses to make better predictions, independent from how frequently page faults occur.

We propose `HoPP`, a **H**ardware-software c**o**-designed **P**age **P**refetching framework, which acts as a separate data plane complementing existing kernel-based remote systems such as Fastswap. Specifically, ① in hardware, we introduce two hardware modules in MC, *i.e.*, hot page detection and reverse page table cache. They take the Last Level Cache (LLC) misses as input and output a sequence of *ordered, real-time* page-based memory trace to `HoPP` software. This key factor enables the following major designs in software: ② `HoPP` can run different prefetch algorithms according to observed memory access patterns. We validate that full memory trace not only improves the existing state-of-the-art page prefetching algorithm (*i.e.*, majority-based prefetching in Leap [38]), but also offers more insights on access patterns, which inspires us to design a more sophisticated prefetching design (Adaptive Three-tier Prefetching) for better performance. ③ Real-time memory trace allows `HoPP` to run prefetching asynchronously as a separate data path, alongside the conventional data path of remote accesses via page fault. It starts to prefetch as soon as the decisions are made, potentially hiding the swap latency from applications. ④ `HoPP` injects PTEs for prefetched pages once they return, before been hit, and can know whether they are hit/missed from the memory trace. This greatly eliminates the overhead of prefetch-hit existed in kernel-based remote systems. In addition, we charge the prefetched pages to the cgroup of the application while Fastswap and Leap did not account for.

Combining ① to ④, it is possible for applications to potentially experience near 0 page faults, and achieve a near-local performance, whenever the prefetching accuracy and coverage are high (*e.g.*, over 90%) (§VI). ⑤ The remote swap latency is volatile, which is affected by remote memory access latency including application memory access speed and network delay. The asynchronous data path in ③ enables fine-grained control and scheduling on prefethcing, thus can timely and dynamically react to latency volatility. `HoPP` takes a first step to have a policy engine with two knobs (prefetching intensity and offset) for tweaking prefetch aggressiveness and timeliness.

Systems		Trans.	Generality	Performance		
				DRAM Cache	Light Data Path	Efficient Prefetch
App-Integrated [52], [53]		×	×	✓	✓	✓
Language-Runtime [66]		✓	×	✓	×	✓
Bus-Extended [33]		✓	✓	×	✓	✓
Kernel	*Others [7], [38]*	✓	✓	✓	×	×
	`HoPP`	✓	✓	✓	✓	✓

TABLE I
Disaggregated Memory System Comparisons.

We implemented a proof-of-concept prototype based on a commodity X86 platform, with an Hardware-based memory tracking tool called HMTT [23], to emulate the trace collection in MC. We then emulate the proposed hardware modules in software, and also implement them in Verilog to verify their feasibility. As mentioned, `HoPP` can be a complement to existing kernel-based systems. We run `HoPP`'s software in a separate data plane along with Fastswap or Leap, and only use it as a backend to access remote data via RDMA.

We evaluate `HoPP` with 15 real-world large in-memory applications including GraphX, K-means, and Bayes running on top of Spark, and C-based applications such as K-means, HPL, NPB, Quicksort (§VI). Our results indicate `HoPP` can predict prefetching with higher accuracy and coverage. When half of their working set is disaggregated, applications running on top of `HoPP` only incur 3.53% slowdown with 99.5% coverage and 99.9% accuracy, a 59% improvement over Fastswap and Leap (§VI). In summary, we make the following contributions:

- We propose hot page detection and reverse page table cache in MC, which delivers real-time hot page access trace to OS with little cost.
- We validate that full memory trace not only improves the state-of-the-art prefetcher, but also enables us to propose a new prefetching design, Adaptive Three-tier Prefetching. This key factor also inspires a set of different designs on prefetching like asynchronous prefetching, early PTE injection and policy engine, in software.
- We implemented a proof-of-concept prototype based on a commodity X86 platform, with an hardware-based memory tracking tool (`HMTT`), and evaluated its performance gain with real-world large in-memory applications.

II. Background and Motivations

A. Remote Memory System Design Spectrum

This section and Table I present a comparison among four major approaches to build such a system.

Kernel-based systems. They rely on the virtual memory subsystem [7], [11], [38] for transparent access to remote memory. As a result, applications can run on them as is. But transparency is not free. Page fault incurs high latency, exceeding network latency, which makes the software stack a bottleneck for accessing remote memory. The following is a detailed breakdown of swap operations triggered by a page fault in most kernel-based systems:

(1) Whenever a page is missed in DRAM, the present bit of the PTE is 0, thus a page fault occurs, resulting a context swtich – about $0.3\mu s$.

(2) Kernel traverses page table and locates the PTE – $0.6\mu s$.

(3) Query swapcache for the missing page. If not found, allocate a new local page and swap entry, insert them to the swapcache – about $0.4\ \mu s$.

(4) Transmit 4 KB page over RDMA – about $4\ \mu s$.

(5) Memory reclaim [3], [4] is triggered when the physical memory is insufficient (since Linux v5.8, this is completed in advance). One reclaim operation uses more than $300\ \mu s$ to reclaim multiple batches of pages, – about 2 to $5\ \mu s$ per page.

(6) Establish PTE and return to user space – about $1\ \mu s$.

The worst case latency takes 8.3 to $11.3\ \mu s$ on critical path. Prefetching is an effective way to amortize this overhead. It reads extra pages along with the missing page into the swapcache. Future page faults on these pages will result in swapcache hits, hence avoid the costly network transmission (step 4). We call them *prefetch-hit*, whose latencies are 4.3 to $7.3\ \mu s$. Since Linux v5.8, step 5 is completed before issuing prefetching. Its overhead is reduced to $2.3\mu s$ now.

Application-integrated systems. These systems require applications to use a new set of primitives to access remote memory [52]–[54], such as specialized APIs and data structures [6], [12], [13], [40], [42], [64], or make annotations in source code [16], [28], [29], [48], [51]. Such primitives operate on objects and expose their semantics (*e.g.*, remotable) to allow applications to explicitly control data movement or *prefetch efficiently*. *They sacrifice transparency and generality for better performance* by replacing VMS with a userspace lightweight data path and giving control back to users.

Language-runtime managed systems. This approach changes the memory management components of the language runtime, such as JVM's garbage collection [66], and *facilitates efficient prefetching* [65], to make them more remote-memory friendly. Nonetheless, they are limited specific languages (*e.g.*, users of JVM), hence has *limited generality*.

Bus extension-based systems. This approach relies on coherent interconnects for remote accessing [11], [17], [33]. Though it is expected to achieve transparency, generality and high performance altogether and attract great attention [20], [33]. However, it has two limitations: 1) it fails to leverage local DRAM caching to mitigate the performance loss from using remote memory; 2) As synchronous load/store remote access can go through multiple hops over fabrics, lasting for a few μs, every outstanding memory access needs to hold at least one hardware resource until the operation is completed [67]. Thus, datacenter operators prefer this approach (*e.g.*, CXL) for memory disaggregation within a rack [33].

HoPP. HoPP seeks to *leverage the benefits of the above four categories*. First, it takes the kernel-based system as the basis model thus achieves the design goals of transparency and generality, and leverages local DRAM cache for remote access, which are *not* easy or even possible for the other three to achieve at once. For high performance achieved by application-integrated systems (*e.g.*, AIFM), we seek a novel hardware-software co-design to mitigate the fundamental limitations in current kernel-based systems – limited knowledge of memory access history and large overhead of prefetch-hit.

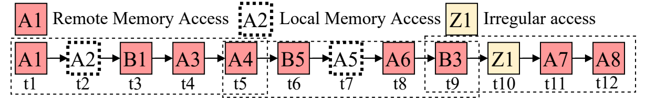

Fig. 1. A motivating example with two streams intertwine, stream A and B have a stride of 2 and 1 respectively. At t_5, Leap cannot derive the dominate stride for stream A due to ① and ②. The same reason for stream B at t_9. Leap cannot derive the stride for stream A at t_{12} due to interference ③.

B. Limited Knowledge of Memory Access History

Existing prefetching algorithms in kernel-based disaggregated systems rely on the missing page history from page faults to identify page streams. A page stream is a sequence of page accesses with a regular stride. However, without memory access to local DRAM, they fail to identify the page streams.

① Page faults produce the missing page addresses, which is *coarse-grained and infrequent*. Thus, a page stream detection will be interrupted and delayed with missing pages only, failing to determine a stable page stride efficiently.

② In highly concurrent scenarios, which is common for large in-memory applications, multiple page streams are accessed alternatively. It further confuses the stride identification with missing pages only, as pages can be from different streams.

③ They fail to filter out *interference pages* that do not belong to any page stream, but falsely take them account into the page streams identification.

Figure 1 showcases an example on how above three limitations impact majority-based prefetching with window 4, the state-of-the-art page prefetcher adopted in Leap [38]. Similarly, strict-pattern prefetcher (*e.g.*, Read-ahead prefetching in Fastswap [7] and Infiniswap [22], VMA-based prefetcher in Linux 5.4) also suffers from the limitations. Intuitively, the above issues can be addressed if full page accesses including both local and remote accesses are provided. We revamp the majority-based prefetcher to utilize full page accesses with two techniques (see details in §III-D): *Pages clustering*: we group pages into different streams based on the fact that streams are separated in different address subspaces, *e.g.*, a new page belongs to a stream if its VPN is within a predefined distance (*e.g.*, 64) from the previously received L pages of that stream; *Large window*: with abundant page access history, each stream adopts a larger window (L) to obtain the dominant stride, which is more robust to interference pages.

We implemented the above prefetch algorithm in HoPP which has full memory trace supply, and compared its prefetching accuracy/coverage to Leap's with microbenchmarks and real applications (detailed setup in (§V)). The result shows that, with full memory access the algorithm improves prefetch accuracy and coverage by 10.6% and by 13.9%, respectively. However, the study on the full memory traces of different applications captured offline, *e.g.*, HPL and NPB-MG (see §VI-D), strongly suggests that there exist another two types of stream patterns – *ladder streams* and *ripple streams*, as shown in Figure 2 and 3, respectively. Thus, the above prefetch algorithm does not fully exploit the full memory trace. In specific, the stream pattern it tries to identify is *simple stream*, *i.e.*, a consecutive page accesses with a fixed stride, but it assumes *there exists only one simple stream within*

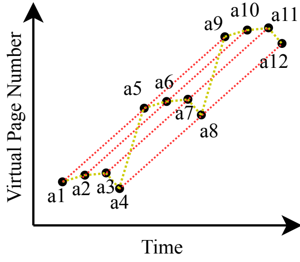

Fig. 2. Ladder streams: Access a1 to a4 forms a ladder tread, and a4 to a5 forms a ladder rise.

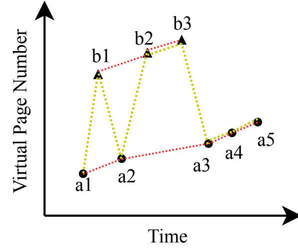

Fig. 3. Ripple streams: Access a1 to a5 forms a ripple stream.

a predefined address subspace, which is strong. In contrast, both ladder and ripple streams are intrinsically formed by simple streams (red lines), their accesses are within a tiny address space (y-axis), and the consecutive page accesses can cross multiple streams in time space (yellow lines). Thus, they cannot be separated with page clustering. Ladder streams have a repetitive spatial access pattern (yellow line in Figure 2), which includes a series of concentrated accesses across streams (ladder tread) followed by a larger, stable stride (ladder rise), *e.g.*, they are common in matrix multiplication's footprint. Ripple streams are a set of special simple streams with stride 1, which are distorted by irregular across-stream and out-of-order accesses.

The existence of different patterns underscores the necessity of full memory trace. As we will show in §III-D, we further exploit the full memory trace, and propose a set of prefetch algorithms that identifies the above three patterns efficiently.

C. Large Overhead of Prefetch-hit.

As mentioned in §II-A, efficient prefetching reduces the overhead of costly remote data path (8.3 to $11.3\mu s$) into the overhead of prefetch-hit ($2.3\mu s$), which is at least 23 times higher than that of a DRAM-hit ($0.1\mu s$). Intuitively, one can eliminate the overhead by simply setting PTEs for prefetched pages once they return – early PTE injection, which is adopted in a kernel-based page prefetching [9] (we call it Depth-N). Although it can turn any prefetch-hit into a DRAM-hit, early PTE injection makes kernel-based remote systems trap into a paradox:

❶ *Limited prefetching flexibility.* Once the PTE for a prefetched page is setup, Depth-N cannot perceive whether it is hit, thus it cannot adjust the algorithm but using a fixed N *e.g.*, 32, while Leap and Fastswap can obtain the prefetching accuracy/coverage from page faults in `Swapcache`.

❷ *Less knowledge of memory accesses.* DRAM-hit cannot trigger any page faults. For systems relying on page faults for prefething, they have less knowledge on the application memory access pattern, further degrading prefetching performance.

❸ *High cost of inaccurate prefetches*. If prefetching is error-prone, PTEs are set for inaccurate prefetches, it would be more difficult to evict them from DRAM, as kernel put it at the very beginning of the LRU-based page list.

As will show in §VI-C, the above problems can offset the performance gain from early PTE injection in Depth-N. As a result, Depth-N performs worse than Fastswap in some cases. Full memory access trace enables another way to tell whether the prefetches are hit or missed thus resolving above issues.

D. Design Space

The fundamental cause of the above two limitations is that OS cannot know where (which memory address) an application is running until page faults. This motivates us to *bridge the gap by decoupling the memory address generation from page faults and exploring a mechanism to collect rich memory trace for prefetching.*

Memory trace collection. Previous works collect memory trace with two approaches. 1) *Software approaches:* The remote memory systems in cloud vendors traverse the access bits of all PTEs periodically, thus tracking the access counts of all pages over a larger period [32], [68]. Based on the access count, they detect hot pages for data migration. Thermostat [5] fires a page fault for a TLB miss and tracks the page's access count according to its misses, which inevitably slows down the application. 2) *Hardware approaches:* They use cacheline-based accesses from hardware components for hot page detection [8], [39], *e.g.*, MMU, MC, *etc.*, and transfer them to OS by setting the PTE's reserved bits. To collect hot pages, the OS also needs to continuously traverse all PTEs to check and reset reserved bits periodically. Similar to software approaches, this memory trace collection is a) *non-real-time* – it is time-consuming to traverse the access bits of a large footprint, *e.g.*, if the average time to check one access bit is $20ns$, the total time to traverse 10GB footprint is $52ms$; b) *costly* – to obtain access count continuously, OS has to check access bits and then reset them periodically. c) *out-of-order* – it reports a collection of the accessed pages without time order. d) *single-bit* – it only tells whether the page is hot.

In contrast, kernel-based prefetching requires memory trace to be 1) *ordered* – prefetched pages should follow the same order; 2) *real-time* – prefetching future pages based on current accessing page improves prefetch coverage (§VI), and 3) *full* – it requires all addresses and timestamps of the page access history to identify access patterns, PID to differentiate applications, shared page flag, *etc.*(§III-C). Thus, the previous trace collections cannot be used for kernel-based prefetching but data migration which operates in a coarse-grained manner and requires 1-bit information only [5], [8], [32], [39], [68].

Why Memory Controller (MC)? The next question is where to collect and process such trace for prefetching. 1) *Memory Management Unit (MMU)*: Since virtual addresses can be collected in MMU, we can directly use them to identify access patterns and determine the pages to prefetch. However, there are four disadvantages. a) MMU sees L1 accesses, which is two orders of magnitude higher than LLC miss (*e.g.*, 180 times for Spark-Graph-BFS). As MMU cannot tell whether the access is in LLC, it will mistakenly consider any page or access pattern with spatial locality inside LLC as a potential stream, which increases hardware complexity for stream identifications and energy cost. b) For each prefetching page, MMU needs to check the present bit of its PTE to ensure it is not in DRAM. This causes the other PTEs to be evicted

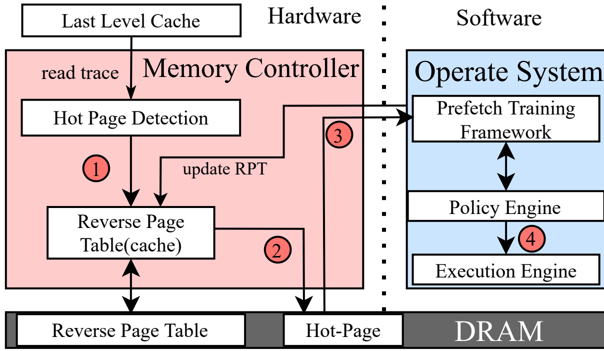

Fig. 4. HoPP Architecture.

from TLB and page table cache at the same core. c) The number of MMUs is much more than MCs, given the in-core resource is more precious, modifying MC is cost-effective. d) When a process migrates between cores, or a page stream from multiple cores, using accesses from a single core cannot identify a complete page stream. 2) *MC*: We choose MC as it processes LLC-misses, which automatically reduces the access volume by filtering out those in-LLC accesses. Kernel-based prefetching cares about large page streams which cannot be cached in DRAM, knowing rich LLC-misses is sufficient to identify large streams [8]. Second, MC belongs to uncore, which is easy to modify than cores and has the least hardware cost. The tradeoff is that we need a mechanism to translate physical-addressed pages to virtual-addressed ones (see PPN-to-VPN mappings in §III-C). 3) *LLC and L2 cache*: LLC also sees LLC-misses, and L2 cache sees more L2-misses. Similar to MMU, the number of LLC banks and L2 cache is much more than MCs, which requires more hardware changes. Additionally, they also need to do PPN-to-VPN mappings.

Full real-time memory access history from MC opens a new design space for kernel-based remote systems in software:
✦ It offers more input knowledge to design a more sophisticated prefetch algorithm with better performance (§III-D);
✦ Prefetching can be designed as a separate data path alongside the conventional data path of remote accesses via page faults, which allows fine-grained control and scheduling of prefetching, and potentially hides the swap latency (§III-E).
✦ Maximizing the benefit of early PTE injection without hurting the prefetching flexibility by calculating the actual prefetching accuracy/ coverage with memory trace (§III-F).

III. HoPP Design

A. Overview

HoPP is a hardware and software co-designed system. We implement a hot page detection (§III-B) and a reverse page table cache (§III-C) in MC. The software part consists of a training module (§III-D), a policy engine (§III-E) and an execution engine (§III-F). Figure 4 presents HoPP's architecture.

HoPP's input is raw memory accesses originated from last-level cache misses. The hot page detection module extracts hot pages from the trace using a small cache. It then forwards the physical address, *i.e.*, physical page number (PPN), of every hot page to the reverse page table cache (step 1). The cache

PPN	Access Number	Send Bit	LRU Bit	
8001	8	1	1	**Repeated detection**
9001	7	0	1	**No Enough Accesses**
7340	8	0	1	**Hot Page Detected**

Fig. 5. Hot Page Detection Table.

in turn maps the PPN of a hot page into its process ID (PID) and virtual page number (VPN), and saves them to a reserved location in DRAM (step 2). Meanwhile, the prefetch training framework will soon use those abundant hot pages to make prefetch decisions (step 3). The policy engine finalizes what pages (including PID and VPN) to fetch and when to fetch them. It then instructs the execution engine to read data from remote using RDMA and to establish page tables (step 4).

B. Hot Page Detection

MC tracks cacheline-based LLC misses. Simply feeding all the raw trace to OS would consume excessive memory bandwidth, and overwhelm HoPP if the software processes every single LLC miss. Additionally, the software needs to filter out interference pages that do not belong to any stream (§III-D), increasing the software complexity and computing resource. On the other hand, *we want to feed as much memory trace as possible to closely resemble the real-time page accesses*. To output memory traces in real-time without consuming excessive memory bandwidth, we add a lightweight Hot Page Detection (HPD) module in MC to convert cacheline-based access into important or hot page-level access trace.

We omit WRITEs but only account for READs for two reasons. 1) Any READ-miss operation immediately generates a READ trace, while a WRITE-miss operation will first generate a READ trace and write the data to the CPU cache, which has a time lag to be evicted and generates a WRITE trace. 2) The RDMA NIC uses DMA write to fetch the pages into local memory from remote, resulting in numerous write accesses. There is no easy way for us to differentiate them from the normal accesses generated by applications.

To identify hot pages, a direct approach is to track the access counts of all physical pages, and then find the ones have been accessed N times within a time window (*e.g.*, $100\mu s$). However, this approach requires non-trivial hardware resources which cannot fit into an MC. In response, we build a small *hot page detection table (HPD table)* to only track the access counts of M pages. In Figure 5, we organize the table as a 16-way 4-set associative cache with LRU replacement ($M = 64$). Each entry in the table records the PPN, number of read accesses, a send bit indicates the entry was identified as a hot page and being extracted, and an LRU bit for replacement. The lowest 2 bits of PPN are used as set index.

We now can describe the whole flow in detail. When the HPD receives a memory access, we convert the cacheline-aligned address into PPN, and locate the table entry using PPN. If not found, we insert it. If found, we check whether the send bit is asserted, if so, we drop this access. Otherwise, we increment the number of accesses. Once the accesses exceeds the threshold N, we extract the PPN and mark it as a hot

N	2	4	8	16	32
K-means	1.72%	1.63%	1.59%	1.56%	1.54%
PageRank	11.72%	4.45%	1.55%	1.07%	0.84%
CC	5.18%	2.16%	1.48%	1.19%	1.02%
LP	3.96%	2.42%	1.84%	1.47%	1.26%
BFS	4.01%	2.36%	1.77%	1.44%	1.23%

TABLE II

THE RATIO BETWEEN HOT PAGES IDENTIFIED AND MEMORY ACCESSES.

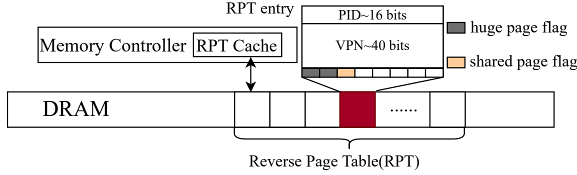

Fig. 6. The only RPT copy resides in DRAM. MC has a 64 KB RPT cache.

page. The more sets in HPD table, the more physical pages can be tracked concurrently. We use four sets, that is, up to 64 different physical pages can be tracked at the same time.

The impact of threshold N. A 4KB page contains 64 cache blocks, thus N ranges from 1 to 64. If N is too small, *e.g.*, $N = 2$, more hot pages are extracted, and the memory trace extracted is closer to the application's real-time page accesses. However, it includes more repeated extraction of the same page, causing more memory bandwidth. Table II shows that when $N < 8$, the number of hot pages extracted increases significantly, resulting in more RPT queries (§III-C), *e.g.*, PageRank, thus the application performance drops (3% for $N = 4$). If N is too large, *e.g.*, $N = 32$, a hot page can be evicted before being accessed N times, and the memory trace extracted is more coarse-grained, which affects the prefetch coverage and application performance. To achieve the best trade-off between memory bandwidth consumption and timely pages extraction, we chose $N = 8$ as default.

The impact of multiple memory channels. When multiple channels are interleaved, different cachelines of the same physical page reside in distinct channels. In this case, we need to reduce N. Although this might lead to repeated hot page extractions, we could de-duplicate them in the prefetch training framework (§III-D). When multiple channels are not interleaved, different hot pages are extracted from different MCs. We can merge them in the prefetch training framework.

C. Reverse Page Table

MC tracks physical addresses and is impossible to determine which physical address belongs to which process. In addition, the cross-page access pattern exists in virtual address space [46]. Thus, we build *Reverse Page Table (RPT)* to map PPN back to PID (*i.e.*, application/process) and VPN. Because it is difficult to store all RPT entries in MC, RPT is stored in a reserved DRAM area. To reduce CPU cache pollution caused by frequent RPT maintenance (covered below), we set RPT to be *uncached*. As shown in Figure 6, each RPT entry has a PID (16 bits), a VPN (40 bits), a shared page flag (1 bit), and a huge page flag (2 bits), total 64 bits. Even with the 4KB page, the whole RPT consumes only 0.17% of the physical memory, *e.g.*, 64 GB local memory requires a 112 MB RPT.

RPT Cache. To reduce the extra memory bandwidth consumed by frequent accesses to RPT in DRAM, we add a small

Size(KB)	1	2	4	8	16	32	64
K-means	0.92	0.93	0.94	0.96	0.98	0.996	0.998
PgRank	0.85	0.86	0.89	0.92	0.97	0.992	0.997

TABLE III

RPT CACHE HIT RATE WHEN VARYING ITS CACHE SIZE.

PID	VPN_history							stride_history						LRU Bit	
231	10	12	14	16	18	20	22	2	2	2	2	2	2	1	→ Stride Detect
231	121	145	171	131	79	132	153	24	26	-40	-52	53	21	1	→ No Stride
230	1	3	5					2	2					1	→ No Enough Pages

Fig. 7. Stream Training Table . A stream is identified when the *VPN_history* is full while the dominant stride has occurred more than $L/2$ times.

RPT cache in MC. All operations made to RPT interact with the cache only, thus there is no need to maintain consistency between the cache and the RPT in DRAM. In other words, all RPT reads and writes pass through this RPT cache inside MC, which ensures consistency. We design RPT cache in 16-way, which takes the PPN emitted from HPD (§III-B) as input and outputs PID+VPN combo, Then, HoPP writes the combo into another reserved DRAM area (step 2 in Figure 4).

RPT Maintenance. When HoPP first starts, it traverses all existing page tables, builds the mappings from PPN to the PID+VPN combo, and then saves the mappings to the RPT in memory. In addition, HoPP installs several hooks into kernel virtual memory subsystem functions. Whenever kernel adds, removes, or updates a PTE, HoPP will be notified and will update the RPT cache accordingly. RPT in memory is updated lazily only when RPT cache writes back dirty entries (§V). Table III evaluates the RPT cache hit rate with various sizes using offline memory trace. With a 64 KB cache, only 0.3% requests miss RPT cache. We see a diminishing return as we further double the cache size (less than 0.1%). Thus, we use 64 KB as our default RPT cache size. The RPT cache hit rate is high because when a page is accessed, it is likely that page was just fetched from remote memory and its page table entry has been established, so its RPT entry exists in the RPT cache.

Shared Page and Huge Page Support. HoPP supports page sharing and huge pages (*e.g.*, 2 MB and 1 GB). As the RPT is indexed by PPN, we support huge pages using the the same hardware infrastructure. Each RPT entry has a huge page flag and a shared page flag. They are not consumed by the RPT module, rather, they are forwarded to the hot page area in memory. It is up to the software to use this information for better predictions (§III-D).

D. Prefetch Training Framework

Prefetch training framework in software can flexible run and update different prefetch algorithms. Rich memory trace from MC enables more vision on access patterns, *i.e.*, simple stream, ladder and ripple (§II-B), thus offers a larger design space to design a more sophisticated prefetch algorithm. We propose Adaptive Three-Tier Prefetching design. Each tier employs a prefetch algorithm to identify one of the patterns and prefetch accordingly. Our proposal is just one solution in a large design space, advanced solutions like machine learning-based ones [58] can also be enabled by full trace.

1) Framework: The framework is responsible to group hot pages into page streams. Then, a prefetch algorithm is applied to look for repetitive patterns inside that stream for prediction.

Algorithm 1 Ladder-Stream-based Prefetch Algorithm

Input: VPN_A, $stride_A$, PID_A, $VPN_history[L]$, $stride_history[L-1]$;
Output: *stride_target*, *pattern_stride* ;

1: $pattern_target[0] = stride_history[L-2]$
2: $pattern_target[1] = stride_A$
3: $next_stride[]$ % to store *stride_target* candidates
4: $stride_sum[]$ % to store *pattern_stride* candidates
5: $last_index = L-2$
6: **for** i in $[L-3$ to $0]$ **do**
7: **if** $stride_history[i] == pattern_target[0]$ and $stride_history[i+1] == pattern_target[1]$ **then**
8: save $stride_history[i+2]$ into $next_stride[]$
9: save $VPN_history[last_index]$ − *VPN_history[i]* into $stride_sum[]$
10: $last_index = i$
11: **if** *next_stride* is not empty **then**
12: *stride_target* = the stride in $next_stride[]$ with the most occurrences
13: *pattern_stride* = the stride in $stride_sum[]$ with the most occurrences
14: **else**
15: $stride_target = 0$, $pattern_stride = 0$
16: send $\{VPN_A$+*stride_target* + $i *$ *pattern_stride*, $PID_A\}$ to prefetch execution engine.

The core of the framework is a *Stream Training Table (STT)*, with 64 entries to identify stream patterns. Figure 7 shows its structure. Each entry represents a potential stream, with a PID field, an LRU bit, a *VPN_history* saving the last L received VPNs with the same PID, and a *stride_history* saving the $L-1$ corresponding strides derived from *VPN_history*. A larger L means a more stringent condition to identify a stream and is more robust to filter out irregular access sequences. We set $L = 16$ in our implementation.

For each hot page A (*i.e.*, VPN_A, PID_A), we first check whether VPN_A belongs to an existing stream in the STT by checking these conditions: (a) whether PID=PID_A, (b) the distance between the *VPN_history*$[last_one]$, the last VPN received in that array, and VPN_A is within a predefined value of Δ_{stream} pages (we use Δ_{stream}=64 to find stream patterns as many as possible). If all conditions are met, we append the new VPN_A to *VPN_history*, and its stride from the previous VPN to *stride_history*.

Once *VPN_history* is full, adaptive three-tier prefetching starts to work: we first apply a Simple-Stream-based Prefetch algorithm (SSP) to identify simple streams as it covers a major part of stream patterns (see §VI-D) and is easy to identify with majority-based detection, If SSP fails, we apply a Ladder-Stream-based Prefetch algorithm (LSP) to identify ladder streams. The last resort is a Ripple-Stream-based Prefetch algorithm (RSP) to identify ripple streams.

2) SSP: We say a stride is dominant in a *stride_history* if a stride value has occurred more than or equal to $L/2$ times. Once a dominant stride is found, we send a prefetch request to the policy engine, whose VPN equals to $VPN_history[L-1] + i \times stride$, where i denotes the prefetch offset (§III-E).

3) LSP: Algorithm 1 shows the detailed algorithm. For a new stride ($stride_A$) in a *stride_history*, LSP identifies *if the stride is part of the repetitive ladder-formed spatial pattern.*

Algorithm 2 Ripple-Stream-based Prefetch Algorithm

Input: VPN_A, $stride_A$, $VPN_history[L]$, $stride_history[L-1]$
Output: *stride_target*;

1: $max_stride = 2$, $ripple_num = 0$, $accumulate_stride = 0$
2: **if** $abs(stride_A) \leq max_stride$ **then**
3: $ripple_num + +$
4: $accumulate_stride = 0$
5: **for** i in $[L-2$ to $0]$ **do**
6: $accumulate_stride$ **+=** $stride_history[i]$
7: **if** $abs(accumulate_stride) \leq max_stride$ **then**
8: $ripple_num + +$
9: $accumulate_stride = 0$
10: **if** $ripple_num \geq L/2$ **then**
11: $stride_target = 1$
12: send { VPN_A + $i *$*stride_target*, PID_A} to prefetch policy engine.

We introduce a smaller target pattern (*pattern_target* in line 1-2), which is a consecutive M strides including $stride_A$, if it is repetitive in *stride_history*, *i.e.*, there exists multiple candidate patterns matched with the target one in *stride_history* (line 6-10), we regard this repetition as a subset of the ladder streams. Thus, the future accesses of the target pattern should follow the spatial correlation between the candidate patterns, which can be derived from *stride_history*. Thus, we determine the next stride of the target pattern (*stride_target*), and the page stride (or distance) between future pattern repetitions (*pattern_stride*) from the *dominant next stride of the candidate patterns* and the *dominant stride between observed repetitive patterns*, respectively. Then, we send a request of prefetching page VPN_A + *stride_target* + $i *$ *pattern_stride* to policy engine. We set $M = 2$, a larger M means a more stringent condition to identify the repetition.

We take Figure 2 as an example. When receiving `a11`, we obtain 2 strides {`a11−a10`, `a10−a9`} to form *pattern_target*, and identify a set of pattern candidates (*pattern_candidate*) matched with *pattern_target* by traversing *stride_history* from its tail (line 6), *i.e.*, {`a7−a6`, `a6−a5`}, {`a3−a2`, `a2−a1`}. For every *pattern_candidate*, we save its next stride into *next_stride*, *i.e.*, `a8−a7` and `a4−a3`, and the page stride from its next pattern candidate or target pattern into *stride_sum*, *i.e.*, `a11−a7` and `a7−a3`, respectively. Thus, *stride_target* and *pattern_stride* are `a8−a7` and `a11−a7`, respectively.

4) RSP: We use RSP as the last resort to identify whether the new stride belongs to a ripple stream. Algorithm 2 shows the detailed algorithm. Recall that, ripple streams intrinsically are a set of simple streams with stride 1, lying within a tiny address space. The insight is that, if a hot page VPN_A belongs to a ripple stream, even if the previous accesses in *VPN_history* hop out of the stream (resulting in a larger stride), there exists an access eventually will be back to the same ripple stream after a few hops, resulting in the absolute cumulative strides from VPN_A equal to 1 (line 2-9). We take Figure 3 as an example. Since the cumulative stride from `a3` to `a2` is 1, `a3` is a ripple page. As a ripple stream can be out-of-order accessed, distorting stride 1, We use max_stride to tolerate out-of-order accesses, which replaces stride 1 to identify ripple pages (line 2 and 7). However,

the across-stream accesses can be mistakenly considered as out-of-order accesses if max_stride is set too large. We set $max_stride = 2$ to allow 2 out-of-order accesses, which happens most of the time. Once the number of ripple pages exceeds half of $VPN_history$ size (line 10), we determine the new page belongs to a ripple stream, thus send a request of prefetching page $VPN_A + i * stride_target$ to policy engine, where *stride_target* = 1 for ripple streams.

E. Prefetch Policy Engine

Obtaining real-time abundant traces from MC enables fine grain control on prefetch behavior, we propose a prefetch policy engine to tune prefetch aggressiveness. The engine has two knobs: *prefetch intensity* and *prefetch offset*. We expect to integrate more policies in the future.

Prefetch intensity In principle, the streams with more intensive memory accesses should prefetch with a higher intensity. With hot pages from MC, HoPP can sense the memory access rate of every stream. To match the memory accessing intensity, HoPP prefetches one page per hot page received, if the corresponding stream is identified. If the current network has abundant bandwidth, HoPP can prefetch more than one page (*e.g.*, 2) to avoid future page faults due to network congestion, as the network bandwidth is too low to sustain the stream's memory access rate.

Prefetch offset (*i.e.*, i in §III-D) ensures timeliness by dictating how far to prefetch when the algorithm has identified a stream pattern for prefetching. For instance, an $i = 1$ means the second page following the stride for ripple streams. We use prefetch offset to ensure that a prefetched page will be ready before it is accessed, thereby avoiding stall from page fault due to uncertainties in OS and network. On the other hand, we should not prefetch too far, which harms the prefetch timeliness, since the page will stay in the local memory for a long time before it is accessed. We denote this time period as T. HoPP measures T for every prefetched page of a stream and makes sure that T is within a predefined interval $[T_{min}, T_{max}]$. If T is smaller than the lower-bound T_{min}, it is likely that the page is going to be late, thus HoPP prefetches further pages by setting $i = i \times (1+\alpha)$. If T is larger than T_{max}, HoPP will prefetch closer pages by setting $i = i \times (1 - \alpha)$. By default, we use $\alpha = 0.2$, $i_{max} = 1K, T_{min} = 40us, T_{max} = 5ms$.

F. Prefetch Execution Engine

The prefetch execution engine is responsible for reading data from the remote and establishing PTEs. In specific, it accepts prefetch requests from the policy engine (§III-E). It checks for duplicated requests and then reads from the remote using RDMA. Whenever receiving a prefetched page, the engine *injects* PTEs to avoid future page faults on this page. To make early PTE injection effective, the prefetch algorithm requires a high accuracy (*e.g.*, over 90%). If the accuracy is low, most of the prefetched pages will not be used. They end up wasting bandwidth and polluting cache. Fortunately, HoPP delivers high prediction accuracy thanks to the full memory access trace (§VI). This explains why HoPP can leverage the early PTE injection technique but Fastswap and Leap cannot.

IV. Discussion

Huge Page Support. HoPP design now supports huge page translation (§III-C). However, kernel-based paging system only supports 4KB page swapping. The kernel swap latency of a 2MB page exceeds $1ms$, which adds more latency in applications' critical path. Thus, it is not desirable to access huge pages remotely. HoPP can be designed to support large page prefetching. First, the kernel needs to reserve 2MB huge page space in advance. When HoPP detects the page stream is long enough, it can choose to swap 512 consecutive future pages with one prefetch request to the reserved 2MB space.

Why hardware-software co-design? The reason why we design HPD and RPT cache in hardware is that they are general functions without frequent updates, minimizing hardware cost, which is easier to add to commercial processors. We implement prefetching training framework in software to support flexible algorithm adaptations or implement a new advanced algorithm like our adaptive three-tier prefetching and machine learning-based ones. In addition, we leave prefetch policy engine in software to adapt to the fluctuated host and network delays. Therefore, HoPP cannot be realized with pure software approach (lack of real-time full memory trace), and pure hardware approach which cannot support flexible algorithms adaptations and new algorithm update. Besides prefetching, the software can serve other purposes with full memory traces, *e.g.*, improving kernel page eviction.

V. Implementation

We implemented a proof-of-concept prototype based on a real commodity X86 platform instead of implementing HoPP in a simulator, because the speed of the simulator is too slow, *e.g.*, Gem5 with full system mode is thousands of times slower than the actual application running time [10], which lead it unable to simulate full application execution. Specifically, we implemented HoPP as a separate data path in Linux kernel v4.11, alongside the data path of a remote memory system preinstalled at the same server and used HMTT that can track and output memory trace, to emulate the memory tracking module in the design (see Figure 4).

In principle, HoPP can work with any existing kernel-based disaggregated memory systems with little modifications. HoPP will not affect other systems' workflow. In specific, the prefetching data path in HoPP is independent of other systems' page fault and swapping path. Nonetheless, HoPP may still share the networking infrastructure with others. Though both are state-of-the-art kernel-based disaggregated memory systems, Fastswap [7] performs better than Leap [38] (§VI). Hence we integrate HoPP with Fastswap.

Hybrid Memory Trace Tool (HMTT). HMTT [23] uses a DIMM-snooping mechanism to monitor the memory bus. HMTT can collect full off-chip memory reference traces originated from applications. Besides, it can correlate the trace with high-level events such as read and write [23]. We deploy HMTT as a bump-in-the-wire between the memory controller and the DRAM chips.

HMTT-based Memory Tracking Emulation. We build a real platform using HMTT to capture and output the full

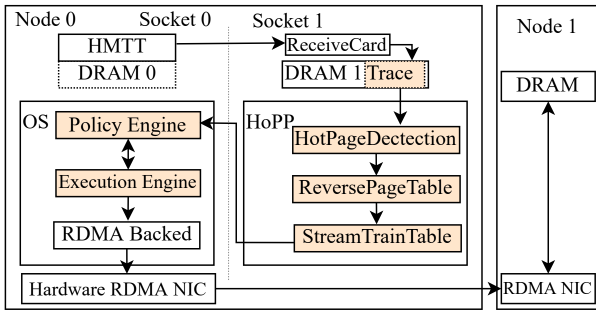

Fig. 8. Reference system architecture of HoPP.

memory access trace (see Figure 8). Originally, HMTT is configured to forward the memory trace collected at one node to the other receiving node via PCIe, which in turn persists the trace on its local SSD [23]. To capture and make use of the trace at the same node, we modified the HMTT configuration. As Figure 8 shows, HMTT is in-between the DIMM and DRAM 0 of Socket 0. It can obtain real-time memory accesses to DRAM 0. By configuring the OS to only run on DRAM 0, HMTT tracks the memory accesses of all running applications. Recall, HoPP hardware design in MC only outputs hot pages, thus it consumes little memory bandwidth, and can write to the same DRAM as applications. However, HMTT outputs all the memory access traces. First, To avoid memory bandwidth interference at the same DRAM, we configure HMTT to write the traces collected in DRAM 0 to DRAM 1 by sending the traces via PCIe to a hardware-based receiving card that is responsible to write them to a reserved area in DRAM 1 with DMA. Each trace has four fields: 8-bit sequence number, 8-bit timestamp, 1 bit read/write flag, and 29-bit physical address. DRAM 0 and DRAM 1 are located in separate sockets, so that the write of memory traces cannot be captured again by HMTT. Second, we have to realize HPD in software. which is different from the design (§III-B). HPD reads traces from that reserved area in DRAM 1 to detect hot pages and forwards them to RPT, thus it takes up an additional CPU core. Note that the rest of the prototype implementation follows the design (§ III).

Reverse Page Table Maintenance We install callbacks to kernel functions to keep the RPT up to date. For instance, we use `set_pte_at` and `pte_clear` for 4 KB pages. Similarly, we use `set_pmd_at` and `pmd_clear` for huge pages. Whenever kernel invokes these functions HoPP will update the RPT accordingly.

VI. Performance Evaluations

In this section, we first evaluate real-world large in-memory applications with HoPP (see Table IV). We compare HoPP to two disaggregated memory systems, Fastswap [7] and Leap [38] and Depth-N [9], a page prefetching using early PTE injection (§II-C). Second, we perform sensitivity tests for the techniques proposed in HoPP. Finally, we verify the feasibility of our hardware modules.

Testbed. We used two nodes connected to an Infiniband switch. The first node acts as a compute node running application workloads on top of HoPP while the second node provides remote memory. Both servers have 14-cores and a 56 Gbps RDMA NIC. The compute node has 2×32 GB of DRAM while the memory node has 6×8 GB of DRAM. Both HoPP and the other systems under comparison are configured with their default parameters unless specified.

Workloads	Footprint (GB)	Cores
GraphX(BFS,CC,PR,LP)	33	14
Spark-Bayes	33	4
Spark-K-Means	13	3
OMP-K-Means	3.2	2
High Performance Linpack	1.2	2
NPB(CG,FT,LU,MG,IS)	$1-7$	2
QuickSort	4	1

TABLE IV
Application workloads.

Workloads. Table IV shows the 15 real-world large in-memory applications used in our evaluation. Note that the footprints of the Spark applications increase gradually, *e.g.*, the GraphX workload (running with Spark) consumes 33 GB in total, but its running time can be divided into three parts with each part consuming different amount of memory: 11 GB, 22 GB, 33 GB for 1st, 2nd and 3rd part respectively.

A. Metrics

Similar to prior work [38], we measure the performance of prefetching algorithm with three metrics: **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. **Timeliness**: the time gap from the time a prefetched page is received to the time it is first hit. For all tests, we report normalized performance [38]. The baseline is the completion time when a workload is using local memory only. The normalized performance can be calculated as CT_{local}/CT_{system}, where CT_{local} and CT_{system} denote the completion time of the local scenario and the compared disaggregated system, respectively.

B. Real Application Performance

In the section, we evaluate the normalized performances of various workloads in Table IV when running with HoPP. For comparison, we also evaluate the normalized performance of the same workloads with Fastswap and Leap. As Leap performs multiple orders worse than the other two, for clear illustration, the results of Leap are omitted.

Note that JVM-based applications run atop JVM, which can manage the memory configuration differently from the ones without JVM. The consequence is a different memory allocation, *e.g.*, with the same workload, Spark divides Kmeans workload into multiple stages, each stage writes the data into a different memory area, but OMP-Kmeans allocates a large array and writes all the data into a contiguous memory. This leads to more streams patterns in spark applications, and the length of the stream is relatively small, thus the repetitive patterns might stop before HoPP finishes identifying them.

Workloads without JVM. We evaluate the normalized performance of HoPP and Fastswap when running the application workloads with two memory limits, *i.e.*, the local memory is set to 50% and 25% of the workload footprint, respectively. Figure 9 shows that, for 50% memory limit, HoPP's average normalized performance is 67.44%, and 3.53% slowdown

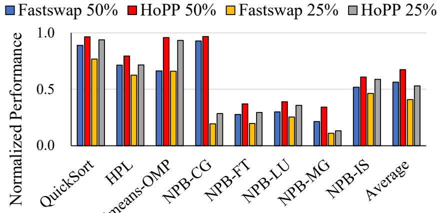

Fig. 9. Normalized performance of Fastswap and HoPP with 50% and 25% local memory using the workloads without JVM.

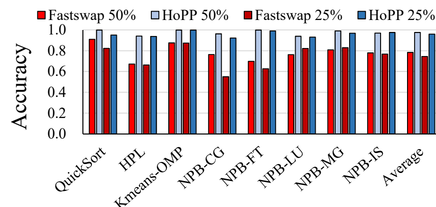

Fig. 10. The accuracy of Fastswap and HoPP's prefetchers running workloads without JVM.

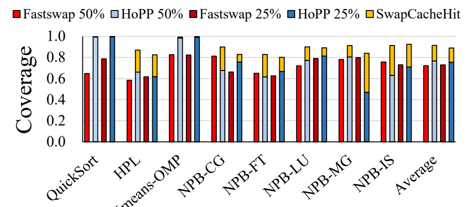

Fig. 11. The coverage of Fastswap and HoPP's prefetching running workloads without JVM.

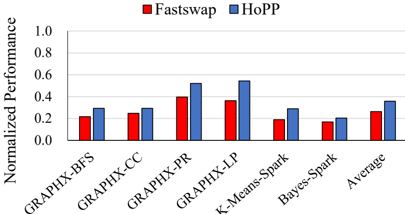

Fig. 12. Normalized performance of Fastswap and HoPP using Spark workloads.

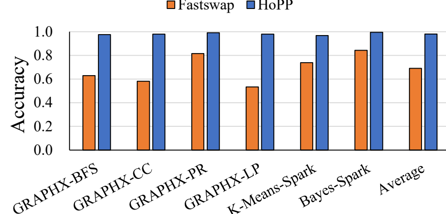

Fig. 13. The accuracy of Fastswap and HoPP's prefetcher running Spark workloads.

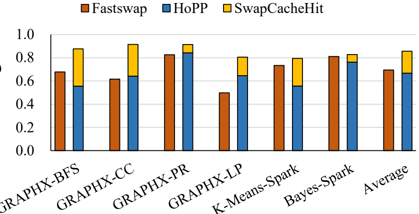

Fig. 14. The coverage of Fastswap and HoPP running Spark workload.

at least, compared to local scenario. For comparison, the average normalized performance of Fastswap is 56.28%, and 11.15% slowdown at least. For 25% memory limit, the average normalized performance of HoPP and Fastswap is 53.07% and 40.91% respectively. The least slowdown for HoPP and Fastswap is 6.20% and 23.24% respectively. Thus, for 50% memory limit, HoPP accelerates Fastswap by 59.8% at most, and 4.2% at least. For 25% memory limit, HoPP accelerates Fastswap by 49.7% at most, 14.7% at least. The average performance improvement over Fastswap is 24.9% and 32%, with 50% and 25% memory limit, respectively.

Figure 10 shows the prefetching accuracy of HoPP and Fastswap's prefetching, respectively. The prefetching accuracy of HoPP is over 90%, implying that almost every prefetch from HoPP is correct, thereby greatly improving the performances of the application workloads (in Figure 9). In addition, there are few incorrect prefetches, thus wasting little network bandwidth and polluting local memory. The average accuracy improvement is 18% over Fastswap's.

Figure 11 shows the coverage of HoPP and Fastswap's prefetching, respectively. The prefetching coverage of HoPP is divided into two parts: one part is the number of the pages prefetched during page faults. Whenever these prefetched pages are accessed by the application, they will cause page faults and hits in Swapcache. Since Fastswap always prefetches upon page faults, its coverage only contains Swapcache-hits as shown in Figure 11. The other part is the number of pages prefetched according to the prefetcher implemented in prefetch framework (*i.e.*, adaptive three-tier prefetching), which does not cause page faults. It is because HoPP injects PTE whenever a prefetching request is finished. Whenever a prefetched page is hit, it causes a DRAM-hit.

HoPP has the best coverage for QuickSort and Kmeans, with more than 99% coverage, thus no page fault observed. Both high accuracy and coverage ensure that HoPP accelerates the application completion time, even achieves the completion time comparable to the local scenario.

Spark workload. Similarly, we evaluate the performances of spark workloads with HoPP and Fastswap, respectively. The local memory used by Spark-Kmeans is limited to 2 GB, whereas the local memory of the other Spark applications is limited to 11 GB. This is because Spark-Kmeans has a much smaller footprint as shown in Table IV. Figure 12 shows that HoPP's average normalized performance is 35.73%, and 45.69% slowdown at least, for comparison, the average normalized performance of Fastswap is 26.37%, and 60.37% slowdown at least. In comparison, HoPP accelerates Fastswap by 34.7% on average. Specifically, HoPP has the largest acceleration on Spark-Kmeans, by 52.2%, while has the smallest acceleration on GRAPHX-CC, by 18.4%.

Figure 13 and Figure 14 show the prefetching accuracy and coverage of their prefetchings, respectively. HoPP identifies fewer stream patterns in the spark workloads due to the JVM's memory management and GC, thus the coverage of Spark workloads is not as high as the other applications without JVM. Despite this, HoPP is still 18% and 29.1% higher than Fastswap on average prefetching accuracy and coverage.

We evaluate the speedup of HoPP compared to Fastswap when running multiple application workloads simultaneously. We limit each application's local memory to 50% of its footprint, respectively, and isolate applications with cgroup [4]. Figure 15 shows that, HoPP improves the performance for the multiple-applications scenarios. This is because the hot page trace contains application semantics, *i.e.*, PID. We can easily train prefetching algorithms according to PID by aggregating hot pages to the same PID.

C. Comparison with Depth-N

We implemented Depth-N prefetching atop of Fastswap according to [9]. Figure 16 shows the normalized performance of Depth-16, Depth-32, Fastswap and HoPP. Depth-16 and Depth-32 don't necessarily outperform Fastswap for real applications, *e.g.*, NPB-MG, while HoPP achieves the best of four. To validate why early PTE injection does not take effect for Depth-N, we summarized in Figure 17 the ratio of the number of remote accesses of four systems to the case

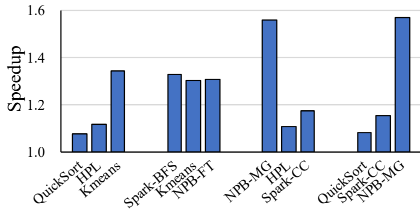

Fig. 15. Speedup when multiple applications run together.

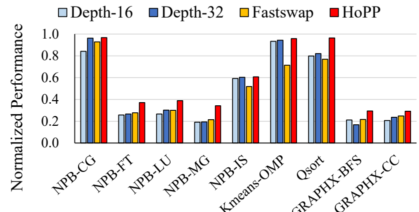

Fig. 16. Normalized performance of Depth-16, Depth-32, Fastswap and HoPP.

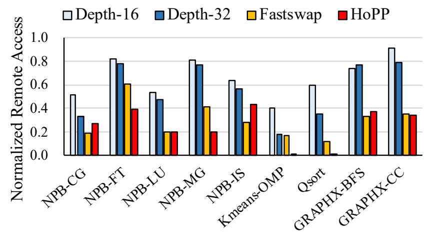

Fig. 17. The ratio of the number of remote memory accesses of Depth-N, Fastswap and HoPP to the one resulted from Fastswap without prefetching.

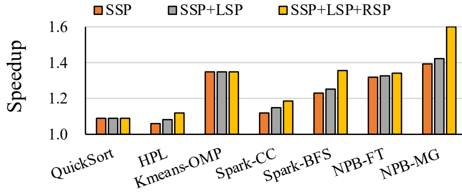

Fig. 18. Speedup of three-tier prefetching.

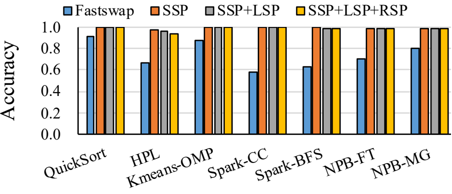

Fig. 19. The accuracy of three-tier prefetching.

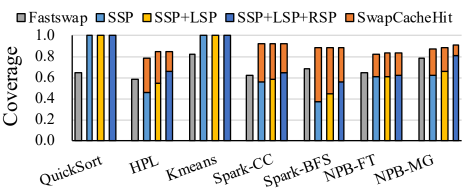

Fig. 20. The coverage of three-tier prefetching.

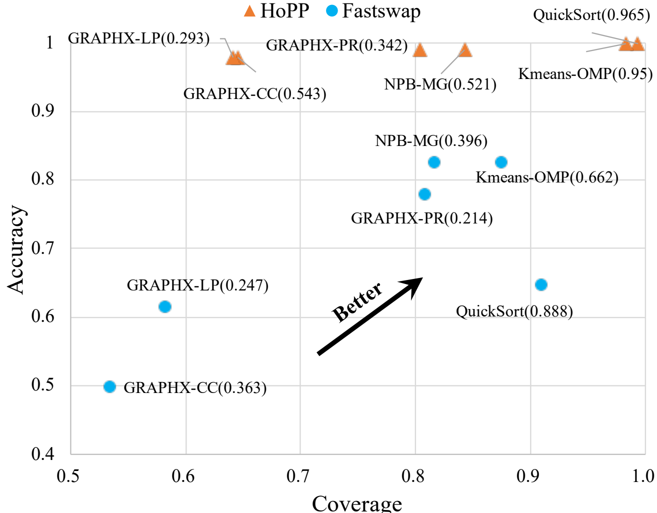

Fig. 21. The impact of accuracy and coverage of the prefetch algorithm on normalized performance. The value in brackets near the application name represents the normalized performance.

without prefetching, *i.e.*, the number of remote accesses when running Fastswap without prefetching (we call it *normalized remote access*). As shown in Figure 17, Depth-N results in the most remote accesses of the four, which shows that its rigid prefetching algorithm cannot effectively reduce page faults when facing various access patterns. Note that, although HoPP does not necessarily have the maximum reduction, it has the best performance, which attributes to early PTE injection without hurting the prefetching flexibility (§II-C).

D. Prefetching Performance Deep Dive

Figure 19 and Figure 20 show the accuracy and coverage improvement by the three prefetch algorithms, *i.e.*, SSP, LSP, RSP, in adaptive three-tier prefetching (§III-D). The accuracy of each algorithm is high (over 90%), as combining them together does no reduce the accuracy. For coverage, simple streams identified by SSP take a major part, while LSP and RSP can further improve the coverage, *e.g.*, for HPL and NPG-MG, LSP offers an additional 9.1% coverage, and RSP can provide an additional 10% coverage.

We use **completion time speedup (Speedup)** to evaluate the performance of every prefetch algorithm. We take Fastswap as the baseline, thus it is defined as $1 - CT_{system}/CT_{Fastswap}$. where CT_{system} and $CT_{Fastswap}$ denote the completion time of the system to compare, and Fastswap, respectively. Figure 18 shows, with more algorithms added, HoPP has a better Speedup. This is because more algorithms improves the prefetch coverage, while still maintains a high prefetch accuracy.

Figure 21 shows the relationship between normalized performance, prefetching accuracy, and prefetching coverage. Note that the coverage of HoPP here only counts the DRAM-hits. For HoPP, if the accuracy and coverage are both close to 1, the normalized performance is approaching to the optimal, 1, regardless of how much the working set is disaggregated. *e.g.*, Quicksort and Kmeans-OMP. This is because HoPP uses early PTE injection and asynchronous prefetching data path to eliminate page faults, thus the application no longer hangs upon page faults. In contrast, Fastswap's accuracy and coverage are worse due to its limited knowledge of memory access history. Interestingly, even if Fastswap's coverage is similar to or better than HoPP's (while both accuracies are similar), Fastswap's application performance is still worse. This is due to the large overhead of prefetch-hit (see §II-C), *e.g.*, for GRAPHX-PR and NPB-MG, HoPP accelerates Fastswap by about 30%, This is because HoPP greatly reduces the overhead of prefetch-hit with early PTE injection, which compensates for the negative effect of the worse coverage.

E. Design Sensitivity

We investigate the effect of every technique used in HoPP on the overall performance. We use *Speedup* defined in §VI-D to evaluate the effect of early PTE injections (§III-F), three-tier prefetching (§III-D) and prefetch offset control (§III-E). The benchmark allocates 2 GB memory per worker thread, and uses 2 threads with each reading and adding-up all the values of all 8-byte blocks within a page (*i.e.*, 512 additions for a page), which emulates a bigdata computation like Kmeans. The local memory is limited to 1 GB.

Program	Kmeans	quicksort	HPL	CG	FT	LU	MG	IS	PR	CC	BFS	LP	Kmeans(S)	Bayes(S)	Average
HPD	0.19	0.17	0.30	0.19	0.19	0.14	0.12	0.12	0.18	0.12	0.14	0.14	0.09	0.11	0.16
RPT	0.004	0.004	0.007	0.005	0.004	0.003	0.003	0.003	0.004	0.003	0.003	0.003	0.002	0.003	0.004

TABLE V

BANDWIDTH CONSUMED BY EXTRACTING HOT PAGE AND QUERYING THE REVERSE PAGE TABLE. (UNIT: %).

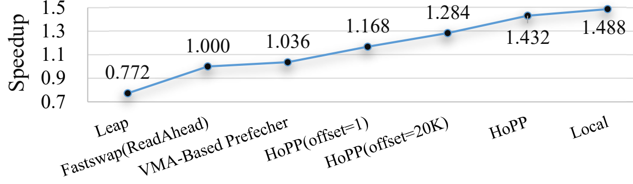

Fig. 22. The impact of different techniques in HoPP, Fastswap is the baseline.

When Leap is used, because two threads run at the same time, its prefetching cannot distinguish different memory access streams, thus calculates the wrong stride, negatively making Leap even worse than Fastswap. VMA-based prefetching is 3.6% better than Fastswap, because it prefetches adjacent pages based on VMA, where VMA is a resemblance of page clustering, but Fastswap prefetches adjacent pages based on swap offset, so VMA-based prefetching prefetches more accurately. HoPP's performance is 40% higher than VMA-based perfeching, which is very close to the one in the local scenario. As this benchmark only contains simple streams with no interference, the pages prefetched by HoPP and VMA-based prefetching are roughly the same. Thus, the 40% of the performance gain is due to early PTE injection which eliminates page faults from happening.

Effect of timeliness. HoPP automatically changes the prefetch offset according to the current timeliness. When HoPP is started, the application must access the remote memory via page faults. The execution speed is very sluggish, thus the timeliness is large. With more prefetch-hits, the timeliness is becoming smaller over time, HoPP will detect it and increase the prefetch offset. As shown in Figure 22, HoPP (with offset adjusted dynamically) performs much better than HoPP with a fixed offset: $i = 1$ (HoPP (offset=1)), and $i = 20K$ (HoPP (offset=20K)), which shows the benefit of prefetch offset control.

F. Hardware Design Feasibility

We implement several modules in Verilog to verify the feasibility of the proposed hardware design together with their memory bandwidth consumption. We leverage CACTI [59] to estimate area and static energy expense using 22 nm technology nodes. (1) **Hot page detection.** We used HMTT to collect offline traces of various applications and analyzed the extra memory bandwidth consumed by writing hot pages. As Table V shows, the average extra bandwidth used by writing hot pages is only 0.16%. This is because at most every N ($N = 8$) accesses results in 1 hot page. The CACTI reports that HPT's area is 0.000252mm^2 and its static energy expanse is 0.0959mw. (2) **Reverse page table.** In §III-C, we analyzed that the hit rate changes after adding different sizes of RPT cache, With a 64KB cache, only about 0.3% of the hot pages are accessed to RPT on DRAM. Table V shows that the average extra bandwidth consumed is only 0.0038%. The CACTI reports that the area is 0.0673 mm^2 and the static energy consumed is $21.4mw$.

VII. RELATED WORK

Remote memory. Many solutions have been proposed to access remote memory, such as using an object-based interface [12], [13], [42], using swapping interface [7], [22], [38], global virtual machine abstraction [66], distributed data stores and file systems [6], [12], [31], [35], [45]. Meanwhile, others propose to use hardware-based methods to access remote memory [36], [44]. HoPP does not rely on page faults and prefetches at any time whenever there is a stream identified.

Prefetch algorithms. A large number of prefetching techniques have been proposed to hide the latency overhead of file accesses and page faults [14], [21], [26]. For cache line granularity prefetching, some works propose to utilize memory-side access patterns [14], [30], [43], [57], [60], [61], injected instructions [15], [28], [29], [48], [51], and hardware features [24], [25], [56], [69]. HoPP utilizes full memory access history to design prefetch algorithms.

VIII. CONCLUSION

This paper presents HoPP, a HW/SW co-designed framework. HoPP introduces hot page detection and reverse page table cache in MC, which transfers sufficient real-time page access trace to the OS at little cost. This key design opens a new design space for kernel-based remote systems. First, it not only improves existing state-of-the-art prefetch algorithms, but also offers more insights to design a more sophisticated prefetch algorithm, *i.e.*, Adaptive Three-tier Prefetching. Second, it allows prefetching to run as a separate data path alongside the conventional data path of remote accesses via page faults, which allows fine-grained prefetching control, and potentially hides the swap latency. Third, it maximizes the benefit of early PTE injection without hurting the prefetching flexibility by calculating the actual prefetching accuracy/coverage using memory trace. We implemented a proof-of-concept prototype based on a commodity X86 platform with a hardware-based memory tracking tool, and evaluated its performance. Experiments show that HoPP performs better than the recent remote systems like Fastswap and Leap.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. This work was supported in part by National Key Research and Development Plan of China (No. 2022YFB4500400), Beijing Municipal Natural Science Foundation (No. 4212028), 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).

REFERENCES

[1] CXL Consortium. https://www.computeexpresslink.org/.

[2] Memcached - a distributed memory object caching system. http://memcached.org.

[3] Page Frame Reclamation. https://www.kernel.org/doc/gorman/html/understand/understand013.html.

[4] Cgroups v2, 2019. https://www.kernel.org/doc/Documentation/cgroup-v2.txt.

[5] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644, 2017.

[6] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, 2018. USENIX Association.

[7] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, 2020.

[8] Apostolos, Kokolis, Dimitrios, Skarlatos, Josep, and Torrellas. Pageseer: Using page walks to trigger page swaps in hybrid memory systems. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[9] Amro Awad, Sergey Blagodurov, and Yan Solihin. Write-aware management of nvm-based memory extensions. In *International Conference on Supercomputing*, pages 1–12, 2016.

[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[11] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA, 2014. USENIX Association.

[13] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. *No Compromises: Distributed Transactions with Consistency, Availability, and Performance*, page 54–70. Association for Computing Machinery, New York, NY, USA, 2015.

[14] Viacheslav Fedorov, Jinchun Kim, Mian Qin, Paul V. Gratz, and A. L. Narasimha Reddy. Speculative paging for future nvm storage. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '17, page 399–410, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. Proactive instruction fetch. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 152–162, New York, NY, USA, 2011. Association for Computing Machinery.

[16] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA, 2016. USENIX Association.

[17] Gen-Z Consortium. https://genzconsortium.org.

[18] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, Hollywood, CA, 2012. USENIX Association.

[19] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, Broomfield, CO, 2014. USENIX Association.

[20] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access,{High-Performance} memory disaggregation with {DirectCXL}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, 2022.

[21] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, page 13, USA, 1994. USENIX Association.

[22] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA, 2017. USENIX Association.

[23] Yongbing Huang, Licheng Chen, Zehan Cui, Yuan Ruan, Yungang Bao, Mingyu Chen, and Ninghui Sun. Hmtt: A hybrid hardware/software tracing system for bridging the dram access trace's semantic gap. *ACM Trans. Archit. Code Optim.*, 11(1), 2014.

[24] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 247–259, New York, NY, USA, 2013. Association for Computing Machinery.

[25] D. Joseph and D. Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121–133, 1999.

[26] Scott F. Kaplan, Lyle A. McGeoch, and Megan F. Cole. Adaptive caching for demand prepaging. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, page 114–126, New York, NY, USA, 2002. Association for Computing Machinery.

[27] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. Rack-scale disaggregated cloud data centers: The dredbox project vision. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 690–695, 2016.

[28] Muneeb Khan, Andreas Sandberg, and Erik Hagersten. A case for resource efficient prefetching in multicores. In *2014 43rd International Conference on Parallel Processing*, 2014.

[29] Aasheesh Kolli, Ali Saidi, and Thomas F. Wenisch. Rdip: Return-address-stack directed instruction prefetching. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 260–271, 2013.

[30] Vamsee Reddy Kommareddy, Jagadish Kotra, Clayton Hughes, Simon David Hammond, and Amro Awad. Prefam: Understanding the impact of prefetching in fabric-attached memory architectures. In *MEMSYS 2020: The International Symposium on Memory Systems*, 2020.

[31] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, New York, NY, USA, 2017. Association for Computing Machinery.

[32] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark Hill, Marcus Fontoura, et al. First-generation memory disaggregation for cloud platforms. *arXiv preprint arXiv:2203.00241*, 2022.

[34] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. Swapping to remote memory over infiniband: An approach using a high performance network block device. In *2005 IEEE International Conference on Cluster Computing*, 2005.

[35] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA, 2014. USENIX Association.

[36] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, Austin, Texas, 2009.

[37] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892, 2017.

[38] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 2020.

[39] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *High-Performance Computer Architecture*, 2015.

[40] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, San Jose, CA, 2013. USENIX Association.

[41] V. Nagarajan, D. J. Sorin, MD Hill, and D. A. Wood. A primer on memory consistency and cache coherence, second edition. *Synthesis Lectures on Computer Architecture*, 15(1):1–294, 2020.

[42] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, Santa Clara, CA, 2015. USENIX Association.

[43] K.J. Nesbit and J.E. Smith. Data cache prefetching using a global history buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 96–96, 2004.

[44] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out numa. *SIGPLAN Not.*, 49(4):3–18, February 2014.

[45] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, New York, NY, USA, 2011. Association for Computing Machinery.

[46] Haiyang Pan, Yuhang Liu, Tianyue Lu, and Mingyu Chen. Lsp: Collective cross-page prefetching for nvm. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 501–506, 2021.

[47] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *ACM*, pages 348–354, 1984.

[48] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. Semantic locality and context-based prefetching using reinforcement learning. *SIGARCH Comput. Archit. News*, 43, 2015.

[49] Peter X. Gao and Akshay Narayan and Sagar Karandikar and Joao Carreira and Sangjin Han and Rachit Agarwal and Sylvia Ratnasamy and Scott Shenker. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savanah, GA, November 2016.

[50] Matthew Poremba, Itir Akgun, Jieming Yin, Onur Kayiran, Yuan Xie, and Gabriel H. Loh. There and back again: Optimizing the interconnect in networks of memory cubes. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, 2017.

[51] Rodric M. Rabbah, Hariharan Sandanagobalane, Mongkol Ekpanyapong, and Weng-Fai Wong. Compiler orchestrated prefetching via speculation and predication. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, page 189–198, New York, NY, USA, 2004. Association for Computing Machinery.

[52] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020.

[53] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, Cambridge, Massachusetts, USA, 1996.

[54] Ioannis T Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, James Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. *ACM SIGPLAN Notices*, 1994.

[55] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.

[56] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, 2000.

[57] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. Efficiently prefetching complex address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 141–152, New York, NY, USA, 2015. Association for Computing Machinery.

[58] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. A hierarchical neural model of data prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 861–873, 2021.

[59] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. *wrl research report*, 2001.

[60] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 69–80, New York, NY, USA, 2009. Association for Computing Machinery.

[61] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74, 2007.

[62] M. Stonebraker and Ariel Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 2013.

[63] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[64] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, New York, NY, USA, 2017. Association for Computing Machinery.

[65] C. Wang, Y. Qiao, H. Ma, S. Liu, Y. Zhang, W. Chen, R. Netravali, M. Kim, and G. H. Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. 2022.

[66] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020.

[67] Luming Wang, Xu Zhang, Tianyue Lu, and Mingyu Chen. Asynchronous memory access unit for general purpose processors. *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, 2(2):100061, 2022.

[68] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, M. Jain, Chunqiang Tang, and Dimitrios Skarlatos. Tmo: transparent memory offloading in datacenters. *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[69] Huaiyu Zhu, Yong Chen, and Xian-He Sun. Timing local streams: Improving timeliness in data prefetching. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, page 169–178, New York, NY, USA, 2010. Association for Computing Machinery.