Effectively Prefetching Remote Memory with Leap

Hasan Al Maruf
University of Michigan

Mosharaf Chowdhury
University of Michigan

Abstract

Memory disaggregation over RDMA can improve the performance of memory-constrained applications by replacing disk swapping with remote memory accesses. However, state-of-the-art memory disaggregation solutions still use data path components designed for slow disks. As a result, applications experience remote memory access latency significantly higher than that of the underlying low-latency network, which itself can be too high for many applications.

In this paper, we propose Leap, a prefetching solution for remote memory accesses due to memory disaggregation. At its core, Leap employs an online, majority-based prefetching algorithm, which increases the page cache hit rate. We complement it with a lightweight and efficient data path in the kernel that isolates each application's data path to the disaggregated memory and mitigates latency bottlenecks arising from legacy throughput-optimizing operations. Integration of Leap in the Linux kernel improves the median and tail remote page access latencies of memory-bound applications by up to $104.04\times$ and $22.62\times$, respectively, over the default data path. This leads to up to $10.16\times$ performance improvements for applications using disaggregated memory in comparison to the state-of-the-art solutions.

1 Introduction

Modern data-intensive applications [5, 29, 30, 70] experience significant performance loss when their complete working sets do not fit into the main memory. At the same time, despite significant and disproportionate memory underutilization in large clusters [62, 78], memory cannot be accessed beyond machine boundaries. Such unused, stranded memory can be leveraged by forming a cluster-wide logical memory pool via *memory disaggregation*, improving application-level performance and overall cluster resource utilization [11, 45, 48].

Two broad avenues have emerged in recent years to expose remote memory to memory-intensive applications. The first requires redesigning applications from the ground up using RDMA primitives [15, 22, 36, 49, 59, 63, 77]. Despite its efficiency, rewriting applications can be cumbersome and may not even be possible for many applications [10]. Alternatives rely on well-known abstractions to expose remote memory; e.g., distributed virtual file system (VFS) for remote file access [10] and distributed virtual memory management (VMM) for *remote memory paging* [28, 32, 45, 46, 65].

Because disaggregated remote memory is slower, keeping hot pages in the faster local memory ensures better performance. Colder pages are moved to the far/remote memory as needed [9, 32, 45]. Subsequent accesses to those cold pages go through a slow data path inside the kernel – for instance, our measurements show that an average 4KB remote page access takes close to 40 μs in state-of-the-art memory disaggregation systems like Infiniswap. Such high access latency significantly affects performance because memory-intensive applications can tolerate at most single μs latency [28, 45]. Note that the latency of existing systems is many times more than the 4.3 μs average latency of a 4KB RDMA operation, which itself can be too high for some applications.

In this paper, we take the following position: *an ideal solution should minimize remote memory accesses in its critical path as much as possible*. In this case, a *local page cache* can reduce the total number of remote memory accesses – a cache *hit* results in a sub-μs latency, comparable to that of a local page access. An effective prefetcher can proactively bring in correct pages into the cache and increase the cache hit rate.

Unfortunately, existing prefetching algorithms fall short for several reasons. First, they are designed to reduce disk access latency by prefetching sequential disk pages in large batches. Second, they cannot distinguish accesses from different applications. Finally, they cannot quickly adapt to temporal changes in page access patterns within the same process. As a result, being optimistic, they pollute the cache with unnecessary pages. At the same time, due to their rigid pattern detection technique, they often fail to prefetch the required pages into the cache before they are accessed.

In this paper, we propose Leap, an online prefetching solution that minimizes the total number of remote memory accesses in the critical path. Unlike existing prefetching algorithms that rely on strict pattern detection, Leap relies on approximation. Specifically, it builds on the Boyer-Moore majority vote algorithm [17] to efficiently identify remote memory access patterns for each individual process. Relying on an approximate mechanism instead of looking for trends in strictly consecutive accesses makes Leap resilient to short-term irregularities in access patterns (e.g., due to multi-threading). It also allows Leap to perform well by detecting trends only from remote page accesses instead of tracing the full virtual memory footprint of an application, which demands continuous scanning and logging of the hardware access bits of the whole virtual address space and results in high CPU and memory overhead. In addition to identifying
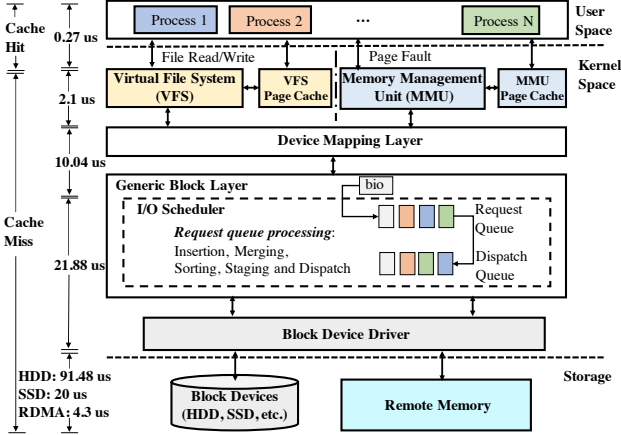
Figure 1: High-level life cycle of page requests in Linux data path along with the average time spent in each stage.

the majority access pattern, Leap determines how many pages to prefetch following that pattern to minimize cache pollution.

While reducing cache pollution and increasing the cache hit rate, Leap also ensures that the host machine faces minimal memory pressure due to the prefetched pages. To move pages from local to remote memory, the kernel needs to scan through the entire memory address-space to find eviction candidates – the more pages it has, the more time it takes to scan. This increases the memory allocation time for new pages. Therefore, alongside a background LRU-based asynchronous page eviction policy, Leap eagerly frees up a prefetched cache just after it gets hit and reduces page allocation wait time.

We complement our algorithm with an efficient data path design for remote memory accesses that is used in case of a cache *miss*. It isolates per-application remote traffic and cuts inessentials in the end-host software stack (e.g., the block layer) to reduce host-side latency and handle a cache miss with latency close to that of the underlying RDMA operations.

Overall, we make the following contributions in this paper:

- We analyze the data path latency overheads for disaggregated memory systems and find that existing data path components can not consistently support single μs 4KB page access latency (§2).

- We propose Leap, a novel online prefetching algorithm (§3) and an eager prefetch cache eviction policy along with a leaner data path, to improve remote I/O latency.

- We implement Leap on Linux Kernel 4.4.125 as a separate data path for remote memory access (§4). Applications can choose either Linux's default data path for traditional usage or Leap for going beyond the machine's boundary using unmodified Linux ABIs.

- We evaluate Leap's effectiveness for different memory disaggregation frameworks. Leap's faster data path and effective cache management improve the median and tail 4KB page access latency by up to 104.04× and 22.62× for micro-benchmarks (§5.1) and by 1.27–10.16× for

real-world memory-intensive applications with production workloads (§5.3).

- We evaluate Leap's prefetcher against practical real-time prefetching techniques (Next-K Line, Stride, Linux Read-ahead) and show that simply replacing the default Linux prefetcher with Leap's prefetcher can provide application-level performance benefit (1.1–3.36× better) even when they are paging to slower storage (e.g., HDD, SSD) (§5.2).

2 Background and Motivation

2.1 Remote Memory

Memory disaggregation systems logically expose unused cluster memory as a global memory pool that is used as the slower memory for machines with extreme memory demand. This improves the performance of memory-intensive applications that have to frequently access slower memory in memory-constrained settings. At the same time, the overall cluster memory usage gets balanced across the machines, decreasing the need for memory over-provisioning per machine.

Access to remote memory over RDMA without significant application rewrites typically relies on two primary mechanisms: disaggregated VFS [10], that exposes remote memory as files and disaggregated VMM for remote memory paging [32, 45, 65]. In both cases, data is communicated in small chunks or pages. In case of remote memory as files, pages go through the file system before they are written to/read from the remote memory. For remote memory paging and distributed OS, page faults cause the virtual memory manager to write pages to and read them from the remote memory.

2.2 Remote Memory Data Path

State-of-the-art memory disaggregation frameworks depend on the existing kernel data path that is optimized for slow disks. Figure 1 depicts the major stages in the life cycle of a page request. Due to slow disk access times – average latencies for HDDs and SSDs range between 4–5 ms and 80–160 μs, respectively – frequent disk accesses have a severe impact on application throughput and latency. Although the recent rise of memory disaggregation is fueled by the hope that RDMA can consistently provide single μs 4KB page access latency [11, 28, 32], this is often a wishful thinking in practice [79]. Blocking on a page access – be it from HDD, SSD, or remote memory – is often unacceptable.

To avoid blocking on I/O, race conditions, and synchronization issues (e.g., accessing a page while the page out process is still in progress), the kernel uses a page cache. To access a page from slower memory, it is first looked up in the appropriate cache location; a *hit* results in almost memory-speed page access latency. However, when the page is not found in the cache (i.e., a *miss*), it is accessed through a costly block device I/O operation that includes several queuing and batching stages to optimize disk throughput by merging multiple contiguous smaller disk I/O requests into a single large re-
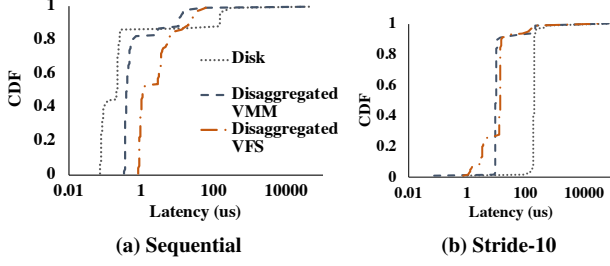
Figure 2: Data path latencies for two access patterns. Memory disaggregation systems have some constant implementation overheads that cap their minimum latency to around 1 μs.

quest. On average, these batching and queuing operations cost around 34 μs and over a few milliseconds at the tail. As a result, a cache miss leads to more than 100× slower latency than a hit; it also introduces high latency variations. For microsecond-latency RDMA environments, this unnecessary wait-time has a severe impact on application performance.

2.3 Prefetching in Linux

Linux tries to store files on the disk in adjacent sectors to increase sequential disk accesses. The same happens for paging. Naturally, existing prefetching mechanisms are designed assuming a sequential data layout. The default Linux prefetcher relies on the last two page faults: if they are for consecutive pages, it brings in several sequential pages into the page cache; otherwise, it assumes that there are no patterns and reduces or stops prefetching. This has several drawbacks. First, whenever it observes two consecutive paging requests for consecutive pages, it over-optimistically brings in pages that may not even be useful. As a result, it wastes I/O bandwidth and causes *cache pollution* by occupying valuable cache space. Second, simply assuming the absence of any pattern based on the last two requests is over-pessimistic. Furthermore, all the applications share the same swap space in Linux; hence, pages from two different processes can share consecutive places in the swap area. An application can also have multiple, inter-leaved stride patterns – for example, due to multiple concurrent threads. Overall, considering only the last two requests to prefetch a batch of pages falter on both respects.

To illustrate this, we measure the page access latency for two memory access patterns: (a) **Sequential** accesses memory pages sequentially, and (b) **Stride-10** accesses memory in strides of 10 pages. In both cases, we use a simple application with its working set size set to 2GB. For disaggregated VMM, it is provided 1GB memory to ensure that 50% of its access cause paging. For disaggregated VFS, it performs 1GB remote write and then another 1GB remote read operations.

Figure 2 shows the latency distributions for 4KB page accesses from disk and disaggregated remote memory for both of the access patterns. For a prefetch size of 8 pages, both perform well for the **Sequential** pattern; this is because 80% of the requests hit the cache. In contrast, we observe signif-
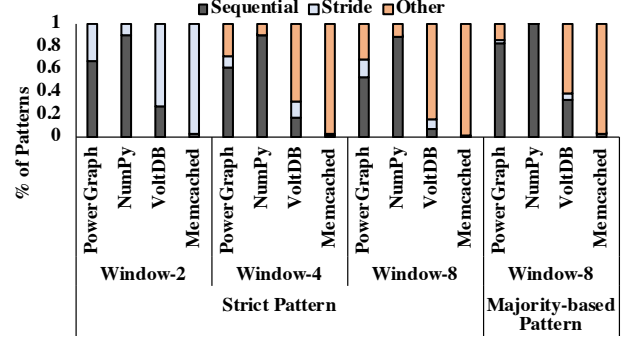

Figure 3: Fractions of sequential, stride, and other access patterns in page fault sequences of length X (Window-X).

icantly higher latency in the **Stride-10** case because all the requests miss the page cache due to the lack of consecutiveness in successive page accesses. By analyzing the latency breakdown inside the data path for **Stride-10** (as shown in Figure 1), we make two key observations. First, although RDMA can provide significantly lower latency than disk (4.3μs vs. 91.5μs), RDMA-based solutions do not benefit as much from that (38.3μs vs. 125.5μs). This is because of the significant data path overhead (on average 34μs) to prepare and batch a request before dispatching it. Significant variations in the preparation and batching stages of the data path cause the average to stray far from the median. Second, the existing sequential data layout-based prefetching mechanism fails to serve the purpose in the presence of diverse remote page access patterns. Solutions based on fixed stride sizes also fall short because stride sizes can vary over time within the same application. Besides, there can be more complicated patterns beyond stride or no repetitions at all.

Shortcoming of Strict Pattern Finding for Prefetching Figure 3 presents the remote page access patterns of four memory-intensive applications during page faults when they are run with 50% of their working sets in memory (more details in Section 5.3). Here, we consider all page fault sequences within a window of size $X \in \{2,4,8\}$ in these applications. Therefore, we divide the page fault scenarios into three categories: *sequential* when all pages within the window of X are sequential pages, *stride* when the pages within the window of X have the same stride from the first page, and *other* when it is neither sequential nor stride.

The default prefetcher in Linux finds strict sequential patterns in window size $X = 2$ and tunes up its aggressiveness accordingly. For example, page faults in PowerGraph and VoltDB follow 67% and 27% sequential pattern within window size $X = 2$, respectively. Consequently, for these two applications, Linux optimistically prefetches many pages into the cache. However, if we look at the $X = 8$ case, the percentage of sequential pages within consecutive page faults goes down to 53% and 7% for PowerGraph and VoltDB, respectively. Meaning, for these two applications, 14–20% of the prefetched pages are not consumed immediately. This creates
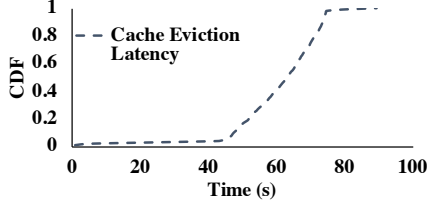
Figure 4: Due to Linux's lazy cache eviction policy, page caches waste the cache area for significant amount of time.

unnecessary memory pressure and might even lead to cache pollution. At the same time, all non-sequential patterns in the $X = 2$ case fall under the stride category. Considering the low cache hit rate, Linux pessimistically decreases/stops prefetching in those cases, which leads to a stale page cache.

Note that strictly expecting all X accesses to follow the same pattern results in not having any patterns at all (e.g., when $X = 8$), because this cannot capture the transient interruptions in sequence. In that case, following the major sequential and/or stride trend within a limited page access history window is more resilient to short-term irregularities. Consecutively, when $X = 8$, a majority-based pattern detection can detect 11.3%–29.7% more sequential accesses. Therefore, it can successfully prefetch more accurate pages into the page cache. Besides sequential and stride access patterns, it is also transparent to irregular access patterns; e.g., for Memcached, it can detect 96.4% of the irregularity.

Prefetch Cache Eviction   Linux kernel maintains an asynchronous background thread (kswapd) to monitor the machine's memory consumption. If the overall memory consumption goes beyond a critical memory pressure or a process's memory usage hits its limit, it determines the eviction candidates by scanning over the in-memory pages to find out the least-recently-used (LRU) ones. Then, it frees up the selected pages from the main memory to allocate new pages. A prefetched cache waits into the LRU list for its turn to get selected for eviction even though it has already been used by a process (Figure 4). Unnecessary pages waiting for eviction in-memory leads to extra scanning time. This extra wait-time due to lazy cache eviction policy adds to the overall latency, especially in a high memory pressure scenario.

3 Remote Memory Prefetching

In this section, we first highlight the characteristics of an ideal prefetcher. Next, we present our proposed online prefetcher along with its different components and the design principles behind them. Finally, we discuss the complexity and correctness of our algorithm.

3.1 Properties of an Ideal Prefetcher

A prefetcher's effectiveness is measured along three axes:

- *Accuracy* refers to the ratio of total cache hits and the total pages added to the cache via prefetching.

- *Coverage* measures the ratio of the total cache hit from the prefetched pages and the total number of requests (e.g., page faults in case of remote memory paging solutions).
- *Timeliness* of an accurately prefetched page is the time gap from when it was prefetched to when it was first hit.

Trade-off   An aggressive prefetcher can hide the slower memory access latency by bringing pages well ahead of the access requests. This might increase the accuracy, but as prefetched pages wait longer to get consumed, this wastes the effective cache and I/O bandwidth. On the other hand, a conservative prefetcher has lower prefetch consumption time and reduces cache and bandwidth contention. However, it has lower coverage and cannot hide memory access latency completely. An effective prefetcher must balance all three.

An effective prefetcher must be adaptive to temporal changes in memory access patterns as well. When there is a predictable access pattern, it should bring pages aggressively. In contrast, during irregular accesses, the prefetch rate should be throttled down to avoid cache pollution.

Prefetching algorithms use prior page access information to predict future access patterns. As such, their effectiveness largely depends on how well they can detect patterns and predict. A real-time prefetcher has to face a trade-off between pattern identification accuracy vs. computational complexity and resource overhead. High CPU usage and memory consumption will negatively impact application performance even though they may help in increasing accuracy.

Common Prefetching Techniques   The most common and simple form of prefetching is spatial pattern detection [51]. Some specific access patterns (i.e., stride, stream, etc.) can be detected with the help of special hardware (HW) features [33, 35, 66, 80]. However, they are typically applied to identify patterns in instruction access that are more regular; in contrast, data access patterns are more irregular. Special prefetch instructions can also be injected into an application's source code, based on compiler or post-execution based analysis [27,40,41,60,61]. However, compiler-injected prefetching needs a static analysis of the cache miss behavior before the application runs. Hence, they are not adaptive to dynamic cache behavior. Finally, HW- or software (SW)-dependent prefetching techniques are limited to the availability of the special HW/SW features and/or application modification.

Summary   An ideal prefetcher should have low computational and memory overhead. It should have high accuracy, coverage, and timeliness to reduce cache pollution; an adaptive prefetch window is imperative to fulfill this requirement. It should also be flexible to both spatial and temporal locality in memory accesses. Finally, HW/SW independence and application transparency make it more generic and robust.

Table 1 compares different prefetching methods.

3.2 Majority Trend-Based Prefetching

Leap has two main components: *detecting trends* and *deter-*

	Low Computational Complexity	Low Memory Overhead	Unmodified Application	HW/SW Independent	Temporal Locality	Spatial Locality	High Prefetch Utilization
Next-N-Line [52]	✓	✓	✓	✓	X	✓	X
Stride [14]	✓	✓	✓	✓	X	✓	X
GHB PC [54]	X	X	✓	X	✓	✓	✓
Instruction Prefetch [27,41]	X	X	X	X	✓	✓	✓
Linux Read-Ahead [72]	✓	✓	✓	✓	✓	✓	X
Leap Prefetcher	✓	✓	✓	✓	✓	✓	✓

Table 1: Comparison of prefetching techniques based on different objectives.

Algorithm 1 Trend Detection

1: **procedure** FINDTREND(N_{split})
2: $\quad H_{size} \leftarrow$ SIZE(*AccessHistory*)
3: $\quad w \leftarrow H_{size}/N_{split}$ ▷ Start with small detection window
4: $\quad \Delta_{maj} \leftarrow \emptyset$
5: $\quad$ **while true do**
6: $\quad\quad \Delta_{maj} \leftarrow$ Boyer-Moore on $\{H_{head}, \ldots, H_{head-w-1}\}$
7: $\quad\quad w \leftarrow w * 2$
8: $\quad\quad$ **if** $\Delta_{maj} \neq$ major trend **then**
9: $\quad\quad\quad \Delta_{maj} \leftarrow \emptyset$
10: $\quad\quad$ **if** $\Delta_{maj} \neq \emptyset$ **or** $w > H_{size}$ **then**
11: $\quad\quad\quad$ **return** Δ_{maj}
12: $\quad$ **return** Δ_{maj}

---

mining what to prefetch. The first component looks for any approximate trend in earlier accesses. Based on the trend availability and prefetch utilization information, the latter component decides how many and which pages to prefetch.

3.2.1 Trend Detection

Existing prefetch solutions rely on strict pattern identification mechanisms (e.g., sequential or stride of fixed size) and fail to ignore temporary irregularities. Instead, we consider a relaxed approach that is robust to short-term irregularities. Specifically, we identify the majority Δ values in a fixed-size (H_{size}) window of remote page accesses (ACCESSHISTORY) and ignore the rest. For a window of size w, a Δ value is said to be the major only if it appears at least $\lfloor w/2 \rfloor + 1$ times within that window. To find the majority Δ, we use the Boyer-Moore majority vote algorithm [17] (Algorithm 1), a linear-time and constant-memory algorithm, over ACCESSHISTORY elements. Given a majority Δ, due to the temporal nature of remote page access events, it can be hypothesized that subsequent Δ values are more likely to be the same as the majority Δ.

Note that if two pages are accessed together, they will be aged and evicted together in the slower memory space at contiguous or nearby addresses. Consequently, the temporal locality in virtual memory accesses will also be observed in the slower page accesses and an approximate stride should be enough to detect that.

Window Management If a memory access sequence follows a regular trend, then the majority Δ is likely to be

(a) at time t_3

(b) at time t_7

(c) at time t_8

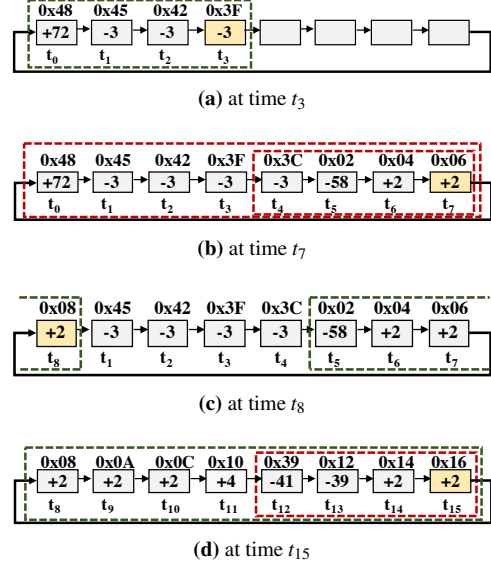
(d) at time t_{15}

Figure 5: Content of ACCESSHISTORY at different time. Solid colored boxes indicate the head position at time t_i. Dashed boxes indicate detection windows. Here, time rolls over at t_8.

found in almost any part of that sequence. In that case, a smaller window can be more effective as it reduces the total number of operations. So instead of considering the entire ACCESSHISTORY, we start with a smaller window that starts from the head position (H_{head}) of ACCESSHISTORY. For a window of size w, we find the major Δ appearing in the $H_{head}, H_{head-1}, \ldots, H_{head-w-1}$ elements.

However, in the presence of short-term irregularities, small windows may not detect a majority. To address this, the prefetcher starts with a small detection window and doubles the window size up to ACCESSHISTORY size until it finds a majority; otherwise, it determines the absence of a majority. The smallest window size can be controlled by N_{split}.

Example Let us consider a ACCESSHISTORY with $H_{size} = 8$ and $N_{split} = 2$. Say pages with the following addresses: 0x48, 0x45, 0x42, 0x3F, 0x3C, 0x02, 0x04, 0x06, 0x08, 0x0A, 0x0C, 0x10, 0x39, 0x12, 0x14, 0x16, were requested in that order. Figure 5 shows the corresponding Δ values stored in ACCESSHISTORY, with t_0 being the earliest and t_{15} being the latest request. At t_i, H_{head} stays at the t_i-th slot.

FINDTREND in Algorithm 1 will initially try to detect a

Algorithm 2 Prefetch Candidate Generation

1: **procedure** GETPREFETCHWINDOWSIZE(page P_t)
2:   PW_{size_t} ▷ Current prefetch window size
3:   $PW_{size_{t-1}}$ ▷ Last prefetch window size
4:   C_{hit} ▷ Prefetched cache hits after last prefetch
5:   **if** $C_{hit} = 0$ **then**
6:     **if** P_t follows the current trend **then**
7:       $PW_{size_t} \leftarrow 1$ ▷ Prefetch a page along trend
8:     **else**
9:       $PW_{size_t} \leftarrow 0$ ▷ Suspend prefetching
10:   **else**   ▷ Earlier prefetches had hits
11:     $PW_{size_t} \leftarrow$ Round up $C_{hit} + 1$ to closest power of 2
12:   $PW_{size_t} \leftarrow \mathbf{min}(PW_{size_t}, PW_{size_{max}})$
13:   **if** $PW_{size_t} < PW_{size_{t-1}}/2$ **then**   ▷ Low cache hit
14:     $PW_{size_t} \leftarrow PW_{size_{t-1}}/2$ ▷ Shrink window smoothly
15:   $C_{hits} \leftarrow 0$
16:   $PW_{size_{t-1}} \leftarrow PW_{size_t}$
17:   **return** PW_{size_t}

18: **procedure** DOPREFETCH(page P_t)
19:   $PW_{size_t} \leftarrow$ GETPREFETCHWINDOWSIZE(P_t)
20:   **if** $PW_{size_t} \neq 0$ **then**
21:     $\Delta_{maj} \leftarrow$ FINDTREND(N_split)
22:     **if** $\Delta_{maj} \neq \emptyset$ **then**
23:       Read PW_{size_t} pages with Δ_{maj} stride from P_t
24:     **else**
25:       Read PW_{size_t} pages around P_t with latest Δ_{maj}
26:   **else**
27:     *Read only page P_t*

---

trend using a window size of 4. Upon failure, it will look for a trend first within a window size of 8.

At time t_3, FINDTREND successfully finds a trend of -3 within the t_0–t_3 window (Figure 5a).

At time t_7, the trend starts to shift from -3 to +2. At that time, t_4–t_7 window does not have a majority Δ, which doubles the window to consider t_0–t_7. This window does not have any majority Δ either (Figure 5b). However, at t_8, a majority Δ of +2 within t_5–t_8 will be adopted as the new trend (Figure 5c).

Similarly, at t_{15}, we have a majority of +2 in the t_8–t_{15}, which will continue to the +2 trend found at t_8 while ignoring the short-term variations at t_{12} and t_{13} (Figure 5d).

3.2.2 Prefetch Candidate Generation

So far we have focused on identifying the presence of a trend. Algorithm 2 determines whether and how to use that trend for prefetching for a request for page P_t.

We determine the prefetch window size (PW_{size_t}) based on the accuracy of prefetches between two consecutive prefetch requests (see GETPREFETCHWINDOWSIZE). Any cache hit of the prefetched data between two consecutive prefetch requests indicates the overall effectiveness of the prefetch. In case of high effectiveness (i.e., a high cache hit), PW_{size_t} is

expanded until it reaches maximum size ($PW_{size_{max}}$). On the other hand, low cache hit indicates low effectiveness; in that case, the prefetch window size gets reduced. However, in the presence of drastic drops, prefetching is not suspended immediately. The prefetch window is shrunk smoothly to make the algorithm flexible to short-term irregularities. When prefetching is suspended, no extra pages are prefetched until a new trend is detected. This is to avoid cache pollution during irregular/unpredictable accesses.

Given a non-zero PW_{size}, the prefetcher brings in PW_{size} pages following the current trend, if any (DOPREFETCH). If no majority trend exists, instead of giving up right away, it speculatively brings PW_{size} pages around P_t's offset following the previous trend. This is to ensure that short-term irregularities cannot completely suspend prefetching.

Prefetching in the Presence of Irregularity FINDTREND can detect a trend within a window of size w in the presence of at most $\lfloor w/2 \rfloor - 1$ irregularities within it. If the window size is too small or the window has multiple perfectly interleaved threads with different strides, FINDTREND will consider it a random pattern. In that case, if the PW_{size} has a non-zero value then it performs a speculative prefetch (line 25) with the previous Δ_{maj}. If that Δ_{maj} is one of the interleaved strides, then this speculation will cause cache hit and continue. Otherwise, PW_{size} will eventually be zero and the prefetcher will stop bringing unnecessary pages. In that case, the prefetcher cannot be worse than the existing prefetch algorithms.

Prefetching During Constrained Bandwidth In Leap, faulted page read and prefetch are done asynchronously. Here, prefetching has a lower priority. In extreme bandwidth constraints, prefetched pages will take a long time to arrive and result in fewer cache hits. This will eventually shrink down PW_{size}. Thus, dynamic prefetch window sizing will help in bandwidth-constrained scenarios.

Effect of Huge Page Linux kernel splits a huge page into 4KB pages before swapping. When transparent huge page is enabled, Leap will be applied on these splitted 4KB pages.

Note that, using huge pages will result in high amplification for dirty data [18]. Besides, average RDMA latencies for 4KB vs 2MB page are 3μs vs 330μs. If huge pages were never split, to maintain single μs latency for 2MB pages, we will need a significantly larger prefetch window size ($PW_{size} \geq 128$), demanding more bandwidth and cache space, and making mispredictions more expensive.

3.3 Analysis

Time Complexity The FINDTREND function in Algorithm 1 initially tries to detect trend aggressively within a smaller window using the Boyer-Moor Majority Voting algorithm. If it fails, then it expands the window size. The Boyer-Moor Majority Voting algorithm (line 6) detects a majority element (if any) in $O(w)$ time, where w is the size of the window. In the worst case, it will invoke the Boyer-Moor
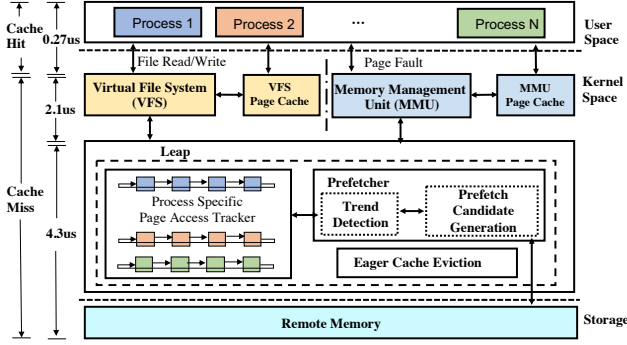
Figure 6: Leap has a faster data path for a cache miss.

Majority Voting algorithm for $O(logH_{size})$ times. However, as the windows are continuous, searching in a new window does not need to start from the beginning and the algorithm never access the same item twice. Hence, the worst-case time complexity of the FINDTREND function is $O(H_{size})$, where H_{size} is the size of the ACCESSHISTORY queue. For smaller H_{size} the computational complexity is constant. Even for $H_{size} = 32$, the prefetcher provides significant performance gain (§5) that greatly outweighs the slight extra computational cost.

Memory Complexity The Boyer-Moor Majority Voting algorithm operates on constant memory space. FINDTREND just invokes the Boyer-Moor Majority Voting algorithm and does not require any additional memory to execute. So, the Trend Detection algorithm needs O(1) space to operate.

Correctness of Trend Detection The correctness of FIND-TREND depends on that of the Boyer-Moor Majority Voting algorithm, which always provides the majority element, if one exists, in linear time (see [17] for the formal proof).

4 System Design

We have implemented our prefetching algorithm as a data path replacement for memory disaggregation frameworks (we refer to this design as Leap data path) alongside the traditional data path in Linux kernel v4.4.125. Leap has three primary components: a page access tracker to isolate processes, a majority-based prefetching algorithm, and an eager cache eviction mechanism. All of them work together in the kernel space to provide a faster data path. Figure 6 shows the basic architecture of Leap's remote memory access mechanism. It takes only around 400 lines of code to implement the page access tracker, prefetcher, and the eager eviction mechanism.

4.1 Page Access Tracker

Leap isolates each process's page access data paths. The page access tracker monitors page accesses inside the kernel that enables the prefetcher to detect application-specific page access trends. Leap does not monitor in-memory pages (hot pages) because continuously scanning and recording the hardware access bits of a large number of pages causes significant computational overhead and memory consumption. Instead,

it monitors only the cache look-ups and records the access sequence of the pages after I/O requests or page faults, trading off a small loss in access pattern detection accuracy for low resource overhead. As temporal locality in the virtual memory space results in a spatial locality in the remote address space, just monitoring the remote page accesses is often enough.

The page access tracker is added as a separate control unit inside the kernel. Upon a page fault, during the page-in operation (`do_swap_page()` under `mm/memory.c`), we notify (`log_access_history()`) Leap's page access tracker about the page fault and the process involved. Leap maintains process-specific fixed-size (H_{size}) FIFO ACCESSHISTORY circular queues to record the page access history. Instead of recording exact page addresses, however, we only store the difference between two consecutive requests (Δ). For example, if page faults happen for addresses `0x2`, `0x5`, `0x4`, `0x6`, `0x1`, `0x9`, then ACCESSHISTORY will store the corresponding Δ values: 0, +3, -1, +2, -5, +8. This reduces the storage space and computation overhead during trend detection (§3.2.1).

4.2 The Prefetcher

To increase the probability of cache hit, Leap incorporates the majority trend-based prefetching algorithm (§3.2). Here, the prefetcher considers each process's earlier remote page access histories available in the respective ACCESSHISTORY to efficiently identify the access behavior of different processes. Because threads of the same process share memory with each other, we choose process-level detection over thread-based. Thread-based pattern detection may result in requesting the same page for prefetch multiple times for different threads.

Two consecutive page access requests are temporally correlated in the sense that they may happen together in the future. The Δ values stored in the ACCESSHISTORY records the spatial locality in the temporally correlated page accesses. Therefore, the prefetcher utilizes both temporal and spatial localities of page accesses to predict future page demand.

The prefetcher is added as a separate control unit inside the kernel. While paging-in, instead of going through the default `swapin_readahead()`, we re-route it through the prefetcher's `do_prefetch()` function. Whenever the prefetcher generates the prefetch candidates, Leap bypasses the expensive request scheduling and batching operations of the block layer (`swap_readpage()`/`swap_writepage()` for paging and `generic_file_read()`/`generic_file_write()` for the file systems) and invokes `leap_remote_io_request()` to re-direct the request through Leap's asynchronous remote I/O interface over RDMA (§4.4).

4.3 Eager Cache Eviction

Leap maintains a circular linked list of prefetched caches (PREFETCHFIFOLRULIST). Whenever a page is fetched from remote memory, besides the kernel's global LRU lists, Leap adds it at the tail of the linked list. After the prefetch cache

gets hit and the page table is updated, Leap marks the page as an eviction candidate. A separate background process continuously removes eviction candidates from PREFETCHFI-FOLRULIST and frees up those pages to the buddy list. As an accurate prefetcher is timely in using the prefetched data, in Leap, prefetched caches do not wait long to be freed up. For workloads where repeated access to paged-in data is not so common, this eager eviction of prefetched pages reduces the wait time to find and allocate new pages - on average, page allocation time is reduced by 750ns (36% less than the usual). Thus, new pages can be brought to the memory more quickly leading to a reduction in the overall data path latency. For workloads where paged-in data is repeatedly used, Leap considers the frequency of access for prefetched pages and exempt them from eager eviction.

However, if the prefetched pages need to be evicted even before they get consumed (e.g., at severe global memory pressure or extreme constrained prefetch cache size scenario), due to the lack of any access history, prefetched pages will follow a FIFO eviction order among themselves from the PREFETCHFIFOLRULIST. Reclamation of other memory (file-backed or anonymous page) follows the existing LRU-based eviction technique by `kswapd` in the kernel. We modify the kernel's Memory Management Unit (`mm/swap_state.c`) to add the prefetch eviction related functions.

4.4 Remote I/O Interface

Similar to existing works [10, 32], Leap uses an agent in each host machine to expose a remote I/O interface to the VFS/VMM over RDMA. The host machine's agent communicates to another remote agent with its resource demand and performs remote memory mapping. The whole remote memory space is logically divided into fixed-size memory slabs. A host agent can map slabs across one or more remote machine(s) according to its resource demand, load balancing, and fault tolerance policies.

The host agent maintains a per CPU core RDMA connection to the remote agent. We use the multi-queue IO queuing mechanism where each CPU core is configured with an individual RDMA dispatch queue for staging remote read/write requests. Upon receiving a remote I/O request, the host generates/retrieves a slot identifier, extracts the remote memory address for the page within that slab, and forwards the request to the RDMA dispatch queue to perform read/write over the RDMA NIC. During the whole process, Leap completely bypasses the expensive block layer operations.

Resilience, Scalability, & Load Balancing One can use existing memory disaggregation frameworks [10, 32, 65] with respective scalability and fault tolerance characteristics and still have the performance benefits of Leap. We do not claim any innovation here. In our implementation, the host agent leverages the power of two choices [53] to minimize memory imbalance across remote machines. Remote in-memory replication is the default fault tolerance mechanism in Leap.

5 Evaluation

We evaluate Leap on a 56 Gbps InfiniBand cluster on Cloud-Lab [3]. Our key results are as follows:

- Leap provides a faster data path to remote memory. Latency for 4KB remote page accesses improves by up to 104.04× (24.96×) at the median and 22.06× (17.32×) at the tail in case of Disaggregated VMM (VFS) (§5.1).
- While paging to disk, our prefetcher outperforms its counterparts (Next-K, Stride, and Read-Ahead) by up to 1.62× for cache pollution and up to 10.47× for cache miss. It improves prefetch coverage by up to 37.51% (§5.2).
- Leap improves the end-to-end application completion times of PowerGraph, NumPy, VoltDB, and Memcached by up to 9.84× and their throughput by up to 10.16× over existing memory disaggregation solutions (§5.3).

Methodology We integrate Leap inside the Linux kernel, both in its VMM and VFS data paths. As a result, we evaluate its impact on three primary mediums.

- *Local disks*: Here, Linux swaps to a local HDD and SSD.
- *Disaggregated VMM (D-VMM)*: To evaluate Leap's benefit for disaggregated VMM system, we integrate Leap with the latest commit of Infiniswap on GitHub [4].
- *Disaggregated VFS (D-VFS)*: To evaluate Leap's benefit for a disaggregated VFS system, we add Leap to our implementation of Remote Regions [10], which is not open-source.

For both of the memory disaggregation systems, we use respective load balancing and fault tolerance mechanisms. Unless otherwise specified, we use ACCESSHISTORY buffer size H_{size} = 32, and maximum prefetch window size $PW_{size_{max}}$ = 8.

Each machine in our evaluation has 64 GB of DRAM and 2× Intel Xeon E5-2650v2 with 16 cores (32 hyperthreads).

5.1 Microbenchmark

We start by analyzing Leap's latency characteristics with the two simple access patterns described in Section 2.

During sequential access, due to prefetching, 80% of the total page requests hit the cache in the default mechanism. On the other hand, during stride access, all prefetched pages brought in by the Linux prefetcher are unused and every page access request experiences a cache miss.

Due to Leap's faster data path, for **Sequential**, it improves the median by 4.07× and 99th percentile by 5.48× for disaggregated VMM (Figure 7a). For **Stride-10**, as the prefetcher can detect strides efficiently, Leap performs almost as good as it does during the sequential accesses. As a result, in terms of 4KB page access latency, Leap improves disaggregated VMM by 104.04× at the median and 22.06× at the tail (Figure 7b).

Leap provides similar performance benefits during memory disaggregation through the file abstraction as well. During sequential access, Leap improves 4KB page access latency by 1.99× at the median and 3.42× at the 99th percentile. During
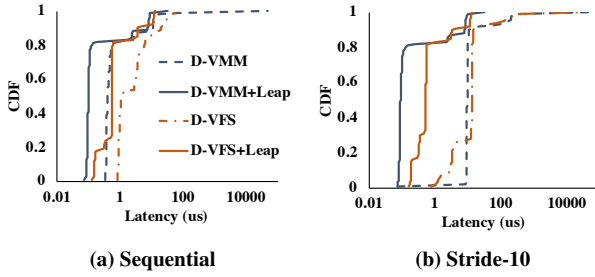
(a) Sequential **(b) Stride-10**

Figure 7: Leap provides lower 4KB page access latency for both sequential and stride access patterns.

stride access, the median and 99th percentile latency improves by 24.96× and 17.32×, respectively.

Performance Benefit Breakdown For disaggregated VMM (VFS), the prefetcher improves the 99th percentile latency by 25.4% (23.1%) over the optimized data path where Leap's eager cache eviction contributes another 9.7% (8.5%) improvement.

As the idea of using far/remote memory for storing cold data is getting more popular these days [9, 32, 45], throughout the rest of the evaluation, we focus only on remote paging through a disaggregated VMM system.

5.2 Performance Benefit of the Prefetcher

Here, we focus on the effectiveness of the prefetcher itself. We use four real-world memory-intensive applications and workload combinations (Figure 3) used in prior works [10, 32].

- TunkRank [8] on PowerGraph [29] to measure the influence of a Twitter user from the follower graph [44]. This workload has a significant amount of stride, sequential, and random access patterns.
- Matrix multiplication on NumPy [57] over matrices of floating points. This has mostly sequential patterns.
- TPC-C benchmark [7] on VoltDB [70] to simulate an order-entry environment. We set 256 warehouses and 8 sites and run 2 million transactions. This has mostly random with a few amount of sequential patterns.
- Facebook's ETC workload [13] on Memcached [5]. We use 10 million SET operations to populate the Memcached server. Then we perform another 10 million queries (5%SETs, 95%GETs). This has mostly random patterns.

The peak memory usage of these applications varies from 9–38.2 GB. To prompt remote paging, we limit an application's memory usage through `cgroups` [2]. To separate the benefit of the prefetcher, we run all of the applications on disk (with existing block layer-based data path) with 50% memory limit.

5.2.1 Prefetch Utilization

We observe the benefit of Leap's prefetcher over following practical and realtime prefetching techniques:

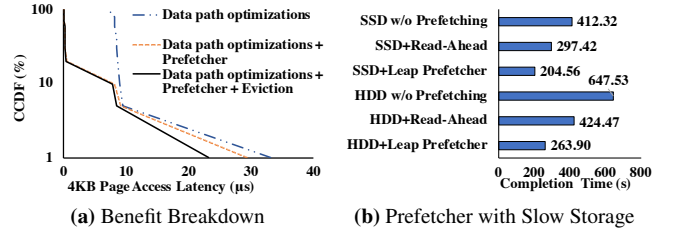- *Next-N-Line Prefetcher* [52] aggressively brings N pages

(a) Benefit Breakdown **(b) Prefetcher with Slow Storage**

Figure 8: The prefetcher is effective for different storage systems.

sequentially mapped to the page with the cache miss if they are not in the cache.

- *Stride Prefetcher* [14] brings pages following a stride pattern relative to the current page upon a cache miss. The aggressiveness of this prefetcher depends on the accuracy of the past prefetch.
- *Linux Read-Ahead* prefetches an aligned block of pages containing the faulted page [72]. Linux uses prefetch hit count and an access history of size 2 to control the aggressiveness of the prefetcher.

Impact on the Cache As the volume of data fetched into the cache increases, the prefetch hit rate increases as well. However, thrashing begins as soon as the working set exceeds the cache capacity. As a result, useful demand-fetched pages are evicted. Table 2 shows that Leap's prefetcher uses fewer page caches (4.37–62.13%) than the other prefetchers for every workload.

A successful prefetcher reduces the number of cache misses by bringing the most accurate pages into the cache. Leap's prefetcher experiences fewer cache miss events (1.1–10.47×) and enhances the effective usage of the cache space.

Application Performance Due to the improvement in cache pollution and reduction of cache miss, using Leap's prefetcher, all of the applications experience the lowest completion time. Based on the access pattern, Leap's prefetcher improves the application completion time by 7.4–75.3% over Linux's default Read-Ahead prefetching technique (Table 2).

Effectiveness If a prefetcher brings every possible page in the page cache, then it will be 100% accurate. However, in reality, one cannot have an infinite cache space due to large data volumes and/or multiple applications running on the same machine. Besides, optimistically bringing pages may create cache contention, which reduces the overall performance.

Leap's prefetcher trades off cache pollution with comparatively lower accuracy. In comparison to other prefetchers, it shows 0.3–10.8% lower accuracy (Table 2). This accuracy loss is linear to the number of cache adds done by the prefetchers. Because the rest of the prefetchers bring in too many pages, their chances of getting lucky hits increase too. Although Leap has the lowest accuracy, its high coverage (0.7–37.5%) allows it to serve with accurate prefetches with a lower cache pollution cost. At the same time, it has an im-

	PowerGraph				NumPy				VoltDB				Memcached			
	Next-N-Line	Stride	Read-Ahead	Leap	Next-N-Line	Stride	Read-Ahead	Leap	Next-N-Line	Stride	Read-Ahead	Leap	Next-N-Line	Stride	Read-Ahead	Leap
Cache Add (millions)	4.88	3.88	3.85	**3.01**	10.75	10.52	10.61	**10.08**	6.50	6.23	5.91	**5.20**	4.65	4.14	4.06	**3.25**
Cache Miss (millions)	1.11	1.61	0.26	**0.15**	0.13	0.16	0.14	**0.12**	1.53	2.24	0.96	**0.90**	1.44	1.39	1.36	**0.96**
Completion Time (s)	683.92	885.86	462.54	**263.90**	1410.30	1380.10	1332.40	**1240.60**	2017.47	2454.72	2064.60	**1799.84**	382.54	374.60	366.91	**302.43**
Accuracy (%)	**55.30**	45.60	45.10	44.60	**89.60**	89.40	89.20	88.90	**40.20**	39.50	39.90	37.60	41.80	**42.10**	41.90	39.40
Coverage (%)	70.90	52.30	86.80	**89.80**	95.80	96.30	96.80	**98.60**	61.20	47.40	68.50	**71.00**	51.70	52.40	56.90	**57.60**
Timeliness (ms) - 95th Percentile	19.10	**0.03**	0.39	0.07	10.34	**0.02**	0.24	0.06	22125.14	**34.32**	64314.96	776.68	32417.89	**466.64**	46679.77	886.67

Table 2: Leap's prefetcher reduces cache pollution and cache miss events. With higher coverage, better timeliness and almost similar accuracy, the prefetcher outperforms its counterparts in terms of application level performance. Here, shaded numbers indicate the best performances.
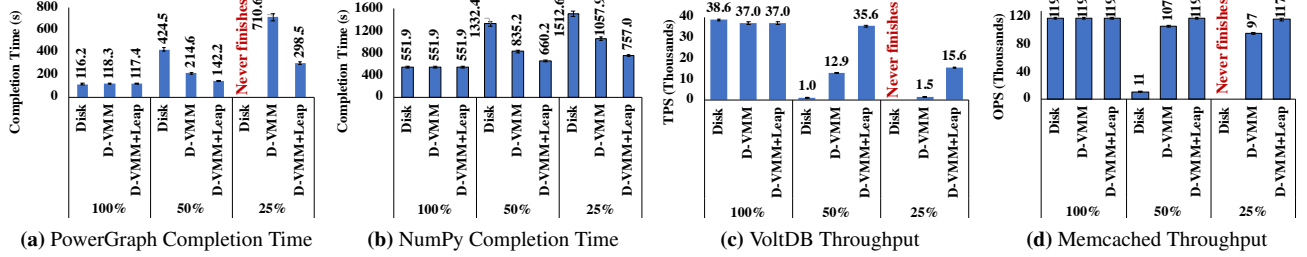

Figure 9: Leap provides lower completion times and higher throughput over Infiniswap's default data path for different memory limits. Note that lower is better for completion time, while higher is better for throughput. Disk refers to HDD in this figure.

proved timeliness over Read-Ahead (4–52.6×) at the 95th percentile. Due to the higher coverage, better timeliness, and almost similar accuracy, Leap's prefetcher thus outperforms others in terms of application-level performance. Note that despite having the best timeliness, Stride has the worst coverage and completion time that impedes its overall performance.

5.2.2 Performance Benefit Breakdown

Figure 8a shows the performance benefit breakdown for each of the components of Leap's data path. For PowerGraph at 50% memory limit, due to data path optimizations, Leap provides with single μs latency for 4KB page accesses up to the 95th percentile. Inclusion of the prefetcher ensures sub-μs 4KB page access latency up to the 85th percentile and improves the 99th percentile latency by 11.4% over Leap's optimized data path. The eager eviction policy reduces the page cache allocation time and improves the tail latency by another 22.2%.

5.2.3 Performance Benefit for HDD and SSD

To observe the usefulness of the prefetcher for different slow storage systems, we incorporate it into Linux's default data path while paging to SSD. For PowerGraph, Leap's prefetcher improves the overall application run time by 1.45× (1.61×) for SSD (HDD) over Linux's default prefetcher (Figure 8b).

5.3 Leap's Overall Impact on Applications

Finally, we evaluate the overall benefit of Leap (including all of its components) for the applications mentioned in Section 5.2. We limit an application's memory usage to fit 100%, 50%, 25% of its peak memory usage. Here, we considered the extreme memory constrain (e.g., 25%) to validate the applicability of Leap to recent resource (memory) disaggre-

gation frameworks that operate on a minimal amount of local memory [65].

PowerGraph PowerGraph suffers significantly for cache misses in Infiniswap (Figure 9a). In contrast, Leap increases the cache hit rate by detecting 19.03% more remote page access patterns over Read-Ahead. The faster the prefetch cache hit happens, the faster the eager cache eviction mechanism frees up page caches and eventually helps in faster page allocations for a new prefetch. Besides, due to more accurate prefetching, Leap reduces the wastage in both cache space and RDMA bandwidth. This improves 4KB remote page access time by 8.17× and 2.19× at the 99th percentile for 50% and 25% cases, respectively. Overall, the integration of Leap to Infiniswap improves the completion time by 1.56× and 2.38× at 50% and 25% cases, respectively.

NumPy Leap can detect most of the remote page access patterns (10.4% better than Linux's default prefetcher). As a result, similar to PowerGraph, for NumPy, Leap improves the completion time by 1.27× and 1.4× for Infiniswap at 50% and 25% memory limit, respectively (Figure 9b). The 4KB page access time improves by 5.28× and 2.88× at the 99th percentile at 50% and 25% cases, respectively.

VoltDB Latency-sensitive applications like VoltDB suffer significantly due to paging. During paging, due to Linux's slower data path, Infiniswap suffers 65.12% and 95.72% lower throughput than local memory behavior at 50% and 25% cases, respectively. In contrast, Leap's better prefetching (11.6% better than Read-Ahead) and instant cache eviction improves the 4KB page access time – 2.51× and 2.7× better 99th percentile at 50% and 25% cases, respectively. However, while executing short random transactions, VoltDB has irregular page access patterns (69% of the total remote page
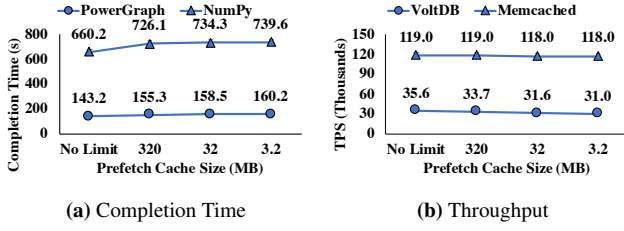
(a) Completion Time   **(b)** Throughput

Figure 10: Leap has minimal performance drop for Infiniswap even in the presence of O(1) MB cache size.

Figure 11: Leap improves application-level performance when all four applications access remote memory concurrently.

accesses). At that time, our prefetcher's adaptive throttling helps the most by not congesting the RDMA. Overall, Leap faces smaller throughput loss (3.78% and 57.97% lower than local memory behavior at 50% and 25% cases, respectively). Leap improves Infiniswap's throughput by $2.76\times$ and $10.16\times$ at 50% and 25% cases, respectively (Figure 9c).

Memcached This workload has a mostly random remote page access pattern. Leap's prefetcher can detect most of them and avoids prefetching in the presence of randomness. This results in fewer remote requests and less cache pollution. As a result, Leap provides Memcached with almost the local memory level behavior at 50% memory limit while the default data path of Infiniswap faces 10.1% throughput loss (Figure 9d). At 25% memory limit, Leap deviates from the local memory throughput behavior by only 1.7%. Here, the default data path of Infiniswap faces 18.49% throughput loss. In this phase, Leap improves Infiniswap's throughput by $1.11\times$ and $1.21\times$ at 50% and 25% memory limits, respectively. Here, Leap provides with $5.94\times$ and $1.08\times$ better 99th percentile 4KB page access time at 50% and 25% cases, respectively.

Performance Under Constrained Cache Size To observe Leap's performance benefit in the presence of limited cache size, we run the four applications in 50% memory limit configuration at different cache limits (Figure 10).

For Memcached, as most of the accesses are of random patterns, most of the performance benefit comes from Leap's faster slow path. For the rest of the applications, as the prefetcher has better timeliness, most of the prefetched caches get used and evicted before the cache size hits the limit. Hence, during O(1) MB cache size, all of these applications face minimal performance drop (11.87–13.05%) compared to the unlimited cache space scenario. Note that, for NumPy, 3.2 MB cache size is only 0.02% of its total remote memory usage.

Multiple Applications Running Together We run all four applications on a single host machine simultaneously with their 50% memory limit and observe the performance benefit of Leap for Infiniswap when multiple throughput- (PowerGraph, NumPy) and latency-sensitive applications (VoltDB, Memcached) concurrently request for remote memory access (Figure 11). As Leap isolates each application's page access path, its prefetcher can consider individual access patterns while making prefetch decisions. Therefore, it brings more
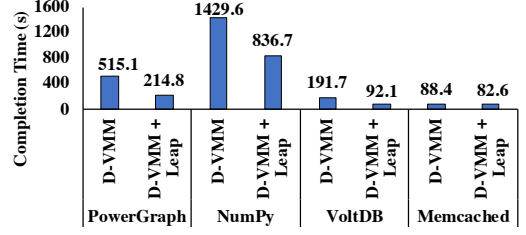
accurate remote pages for each of the applications and reduces the contention over the network. As a result, overall application-level performance improves by 1.1–$2.4\times$ over Infiniswap. To enable aggregate performance comparison, here, we present the end-to-end completion time of application-workload combinations defined earlier; application-specific metrics improve as well.

6 Discussion and Future Work

Thread-specific Prefetching Linux kernels today manage memory address space at the process level. Thread-specific page access tracking requires a significant change in the whole virtual memory subsystem. However, this would help efficiently identify multiple concurrent streams from different threads. Low-overhead, thread-specific page access tracking and prefetching can be an interesting research direction.

Concurrent Disk and Remote I/O Leap's prefetcher can be used for both disaggregated and existing Linux Kernels. Currently, Leap runs as a single memory management module on the host server where paging is allowed through either existing block layers or Leap's remote memory data path. The current implementation does not allow the concurrent use of both block layer and remote memory. Exploring this direction can lead to further benefits for systems using Leap.

Optimized Remote I/O Interface In this work, we focused on augmenting existing memory disaggregation frameworks with a leaner and efficient data path. This allowed us to keep Leap transparent to the remote I/O interface. We believe that exploring the effects of load balancing, fault-tolerance, data locality, and application-specific isolation in remote memory as well as an optimized remote I/O interface are all potential future research directions.

7 Related Work

Remote Memory Solutions A large number of software systems have been proposed over the years to access remote machine's memory for paging [1, 21, 23, 26, 32, 45, 46, 50, 55, 64, 65], global virtual machine abstraction [6, 25, 43], and distributed data stores and file systems [10, 22, 42, 47, 58]. Hardware-based remote access using PCIe interconnects [48] or extended NUMA memory fabric [56] are also proposed to disaggregate memory. Leap is complementary to these works.

Kernel Data Path Optimizations With the emergence of faster storage devices, several optimization techniques, and design principles have been proposed to fully utilize faster hardware. Considering the overhead of the block layer, different service level optimizations and system re-designs have been proposed – examples include parallelism in batching and queuing mechanism [16, 75], avoiding interrupts and context switching during I/O scheduling [12, 20, 74, 76], better buffer cache management [34], etc. During remote memory access, optimization in data path has been proposed through request batching [37, 38, 71], eliminating page migration bottleneck [73], reducing remote I/O bandwidth through compression [45], and network-level block devices [46]. Leap's data path optimizations are inspired by many of them.

Prefetching Algorithms Many prefetching techniques exist to utilize hardware features [33, 35, 66, 80], compiler-injected instructions [27, 40, 41, 60, 61], and memory-side access pattern [24, 54, 67–69] for cache line prefetching. They are often limited to specific access patterns, application behavior, or require specified hardware design. More importantly, they are designed for a lower level memory stack.

A large number of entirely kernel-based prefetching techniques have also been proposed to hide the latency overhead of file accesses and page faults [19, 24, 31, 39, 72]. Among them, Linux Read-Ahead [72] is the most widely used. However, it does not consider the access history to make prefetch decisions. It was also designed for hiding disk seek time. Therefore, its optimistic looking around approach often results in lower cache utilization for remote memory access.

To the best of our knowledge, Leap is the first to consider a fully software-based, kernel-level prefetching technique for DRAM with remote memory as a backing storage over fast RDMA-capable networks.

8 Conclusion

The paper presents Leap, a remote page prefetching algorithm that relies on majority-based pattern detection instead of strict detection. As a result, Leap is resilient to short-term irregularities in page access patterns of multi-threaded applications. We implement Leap in a leaner and faster data path in the Linux kernel for remote memory access over RDMA without any application or hardware modifications.

Our integrations of Leap with two major memory disaggregation systems (namely, Infiniswap and Remote Regions) show that the median and tail remote page access latencies improves by up to $104.04\times$ and $22.62\times$, respectively, over the state-of-the-art. This, in turn, leads to application-level performance improvements of 1.27–$10.16\times$. Finally, Leap's benefits extend beyond disaggregated memory – applying it to HDD and SSD leads to considerable performance benefits as well.

Leap is available at https://github.com/SymbioticLab/leap.

Acknowledgments

References

[1] Accelio based network block device. https://github.com/accelio/NBDX.

[2] cgroups. https://wiki.archlinux.org/index.php/cgroups.

[3] CloudLab. https://www.cloudlab.us.

[4] Infiniswap github repository. https://github.com/SymbioticLab/infiniswap.

[5] Memcached - A distributed memory object caching system. http://memcached.org.

[6] The versatile SMP (vSMP) architecture. http://www.scalemp.com/technology/versatile-smp-vsmp-architecture/.

[7] TPC Benchmark C (TPC-C). http://www.tpc.org/tpcc.

[8] A twitter analog to PageRank. http://thenoisychannel.com/2009/01/13/a-twitter-analog-to-pagerank.

[9] N. Agarwal and T. F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *ASPLOS*, 2017.

[10] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novaković, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: a simple abstraction for remote memory. In *ATC*, 2018.

[11] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *SoCC*, 2017.

[12] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A protoype phase change memory storage array. In *HotStorage*, 2011.

[13] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev.*, 2012.

[14] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *ACM/IEEE Conference on Supercomputing*, 1991.

[15] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using RDMA. In *SIGMOD*, 2015.

[16] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux block io: Introducing multi-queue ssd access on multi-core systems. In *SYSTOR*, 2013.

[17] R. S. Boyer and J. S. Moore. MJRTY: A fast majority vote algorithm. In *Automated Reasoning*. 1991.

[18] I. Calciu, I. Puddu, A. Kolli, A. Nowatzyk, J. Gandhi, O. Mutlu, and P. Subrahmanyam. Project pberry: FPGA acceleration for remote memory. In *HotOS*, 2019.

[19] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *OSDI*, 1994.

[20] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *MICRO*, 2010.

[21] H. Chen, Y. Luo, X. Wang, B. Zhang, Y. Sun, and Z. Wang. A transparent remote paging model for virtual machines. In *International Workshop on Virtualization Technology*, 2008.

[22] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *NSDI*, 2014.

[23] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *IPPS/SPDP*, 1999.

[24] V. Fedorov, J. Kim, M. Qin, P. V. Gratz, and A. L. N. Reddy. Speculative paging for future NVM storage. In *MEMSYS*, 2017.

[25] M. J. Feeley, W. E. Morgan, E. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *SOSP*, 1995.

[26] E. W. Felten and J. Zahorjan. Issues in the implementation of a remote memory paging system. Technical report, University of Washington, 1991.

[27] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive instruction fetch. In *MICRO*, 2011.

[28] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[29] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[30] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[31] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *USTC*, 1994.

[32] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *NSDI*, 2017.

[33] A. Jain and C. Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *MICRO*, 2013.

[34] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. Dulo: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *FAST*, 2005.

[35] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA*, 1997.

[36] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, 2014.

[37] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *ATC*, 2016.

[38] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *OSDI*, 2016.

[39] S. F. Kaplan, L. A. McGeoch, and M. F. Cole. Adaptive caching for demand prepaging. *SIGPLAN Not.*, 2002.

[40] M. Khan, A. Sandberg, and E. Hagersten. A case for resource efficient prefetching in multicores. In *ICPP*, 2014.

[41] A. Kolli, A. Saidi, and T. F. Wenisch. RDIP: Return-address-stack directed instruction prefetching. In *MICRO*, 2013.

[42] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *SOSP*, 2017.

[43] Y. Kuperman, J. Nider, A. Gordon, and D. Tsafrir. Paravirtual Remote I/O. In *ASPLOS*, 2016.

[44] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, 2010.

[45] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. Software-defined far memory in warehouse-scale computers. In *ASPLOS*, 2019.

[46] S. Liang, R. Noronha, and D. K. Panda. Swapping to remote memory over Infiniband: An approach using a high performance network block device. In *Cluster Computing*, 2005.

[47] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *NSDI*, 2014.

[48] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, 2009.

[49] X. Lu, N. S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-performance design of hadoop RPC with RDMA over InfiniBand. In *ICPP*, 2013.

[50] E. P. Markatos and G. Dramitinos. Implementation of a reliable remote memory pager. In *ATC*, 1996.

[51] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Trans. Comput. Syst.*, 1984.

[52] S. Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 2016.

[53] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. *Handbook of Randomized Computing*, 2001.

[54] K. Nesbit and J. Smith. Data cache prefetching using a global history buffer. *IEEE Micro*, 2005.

[55] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A network swapping module for Linux clusters. In *Euro-Par*, 2003.

[56] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *ASPLOS*, 2014.

[57] T. Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006.

[58] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[59] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 2010.

[60] L. Peled, S. Mannor, U. Weiser, and Y. Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *ISCA*, 2015.

[61] R. M. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.-F. Wong. Compiler orchestrated prefetching via speculation and predication. In *ASPLOS*, 2004.

[62] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[63] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. In *PVLDB*, 2015.

[64] A. Samih, R. Wang, C. Maciocco, T.-Y. C. Tai, R. Duan, J. Duan, and Y. Solihin. Evaluating dynamics and bottlenecks of memory collaboration in cluster systems. In *CCGrid*, 2012.

[65] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, 2018.

[66] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *MICRO*, 2000.

[67] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti. Efficiently prefetching complex address patterns. In *MICRO*, 2015.

[68] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. In *ISCA*, 2009.

[69] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.

[70] M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS. *IEEE Data Engineering Bulletin*, 2013.

[71] S.-Y. Tsai and Y. Zhang. Lite kernel rdma support for datacenter applications. In *SOSP*, 2017.

[72] Y. Wiseman, S. Jiang, Y. Wiseman, and S. Jiang. *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*. Information Science Reference - Imprint of: IGI Publishing, 2009.

[73] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee. Nimble page management for tiered memory systems. In *ASPLOS*, 2019.

[74] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *FAST*, 2012.

[75] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Split-level i/o scheduling. In *SOSP*, 2015.

[76] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, H. Eom, and H. Y. Yeom. Optimizing the block i/o subsystem for fast storage devices. *ACM Trans. Comput. Syst.*, 2014.

[77] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The end of a myth: Distributed transactions can scale. In *PVLDB*, 2017.

[78] Q. Zhang, M. F. Zhani, S. Zhang, Q. Zhu, R. Boutaba, and J. L. Hellerstein. Dynamic energy-aware capacity provisioning for cloud computing environments. In *ICAC*, 2012.

[79] Y. Zhang, J. Gu, Y. Lee, M. Chowdhury, and K. G. Shin. Performance Isolation Anomalies in RDMA. In *KBNets*, 2017.

[80] H. Zhu, Y. Chen, and X.-H. Sun. Timing local streams: Improving timeliness in data prefetching. In *ICS*, 2010.