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× and 22.62×, respectively, over the default data path. This leads to up to 10.16× 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 multithreading). 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
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 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 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).

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-
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 100x 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 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 falls 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 significa-

![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.](image)

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

### Table 1: Comparison of Prefetching Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
<th>Coverage</th>
<th>Timeliness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial Pattern Detection</td>
<td>High</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Compiler Injection</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>HW/Software-Dependent</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>

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-
Algorithm 1 Trend Detection

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

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 $\Delta$ values in a fixed-size ($H_{size}$) window of remote page accesses (AccessHistory) and ignore the rest. For a window of size $w$, a $\Delta$ 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 $\Delta$, we use the Boyer-Moore majority vote algorithm [17] (Algorithm 1), a linear-time and constant-memory algorithm, over AccessHistory elements. Given a majority $\Delta$, due to the temporal nature of remote page access events, it can be hypothesized that subsequent $\Delta$ values are more likely to be the same as the majority $\Delta$.

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 $\Delta$ is likely to be 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 $\Delta$ 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: 0x08, 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 $\Delta$ 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 \( P_i \))
2: \( PW_{size_i} \) \( \triangleright \) Current prefetch window size
3: \( PW_{size_{i-1}} \) \( \triangleright \) Last prefetch window size
4: \( C_{hit} \) \( \triangleright \) Prefetched cache hits after last prefetch
5: if \( C_{hit} = 0 \) then
6: if \( P_i \) follows the current trend then
7: \( PW_{size_i} \leftarrow 1 \) \( \triangleright \) Prefetch a page along trend
8: else
9: \( PW_{size_i} \leftarrow 0 \) \( \triangleright \) Suspend prefetching
10: else
11: \( PW_{size_i} \leftarrow \) Round up \( C_{hit} + 1 \) to closest power of 2
12: \( PW_{size_i} \leftarrow \min(PW_{size_i}, PW_{size_{max}}) \)
13: if \( PW_{size_i} < PW_{size_{i-1}} / 2 \) then \( \triangleright \) Low cache hit
14: \( PW_{size_i} \leftarrow PW_{size_{i-1}} / 2 \) \( \triangleright \) Shrink window smoothly
15: \( C_{hit} \leftarrow 0 \)
16: \( PW_{size_{i-1}} \leftarrow PW_{size_i} \)
17: return \( PW_{size_i} \)

18: procedure DO PREFETCH(page \( P_i \))
19: \( PW_{size_i} \leftarrow \) GETPREFETCHWINDOWSIZE(\( P_i \))
20: if \( PW_{size_i} \neq 0 \) then
21: \( \Delta_{maj} \leftarrow \) FINDTREND(\( N_{split} \))
22: if \( \Delta_{maj} \neq 0 \) then
23: Read \( PW_{size_i} \) pages with \( \Delta_{maj} \) stride from \( P_i \)
24: else
25: Read \( PW_{size_i} \) pages around \( P_i \) with latest \( \Delta_{maj} \)
26: else
27: Read only page \( P_i \)

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 \( \Delta \), which doubles the window to consider \( t_0 \)–\( t_7 \). This window does not have any majority \( \Delta \) either (Figure 5b). However, at \( t_8 \), a majority \( \Delta \) of +2 within \( t_8 \)–\( t_{15} \) 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 \( P_i \).

We determine the prefetch window size (\( PW_{size_i} \)) 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), \( PW_{size_i} \) 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 \( PW_{size_i} \), the prefetcher brings in \( PW_{size_i} \) pages following the current trend, if any (DO PREFETCH). If no majority trend exists, instead of giving up right away, it speculatively brings \( PW_{size_i} \) pages around \( P_i \)'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 \( PW_{size_i} \) has a non-zero value then it performs a speculative prefetch (line 25) with the previous \( \Delta_{maj} \). If that \( \Delta_{maj} \) is one of the interleaved strides, then this speculation will cause cache hit and continue. Otherwise, \( PW_{size_i} \) 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 \( PW_{size_i} \). 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 3\( \mu \)s vs 330\( \mu \)s. If huge pages were never split, to maintain single \( \mu \)s latency for 2MB pages, we will need a significantly larger prefetch window size \( PW_{size_i} \geq 128 \), 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
Majority Voting algorithm for $O(\log H_{\text{size}})$ 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 $O(H_{\text{size}})$, where $H_{\text{size}}$ is the size of the ACCESSHISTORY queue. For smaller $H_{\text{size}}$ the computational complexity is constant. Even for $H_{\text{size}} = 32$, 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 FINDTREND 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 ($H_{\text{size}}$) 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 ($\Delta$). For example, if page faults happen for addresses 0x2, 0x5, 0x4, 0x6, 0x1, 0x9, then ACCESSHISTORY will store the corresponding $\Delta$ 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 $\Delta$ 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 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 (PREFIXFIFO/LRU LIST). 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 PREFETCH-FIFO-LRU-LIST and frees up those pages to the buddy list. As an accurate preficher 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 PREFETCH-FIFO-LRU-LIST. 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 preficher 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 ACCESS-HISTORY 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 preficher 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
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].

- **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-

<table>
<thead>
<tr>
<th>4KB Page Access Latency (s)</th>
<th>Data path optimizations</th>
<th>Data path optimizations + Prefetcher</th>
<th>Data path optimizations + Prefetcher + Eviction</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDD+Leap Prefetcher</td>
<td>263.90</td>
<td>324.47</td>
<td>412.32</td>
</tr>
<tr>
<td>HDD+Read-Ahead</td>
<td>297.42</td>
<td>345.36</td>
<td>424.47</td>
</tr>
<tr>
<td>SSD+Leap Prefetcher</td>
<td>263.90</td>
<td>324.47</td>
<td>412.32</td>
</tr>
<tr>
<td>SSD+Read-Ahead</td>
<td>297.42</td>
<td>345.36</td>
<td>424.47</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Completion Time (s)</th>
<th>SSD w/o Prefetching</th>
<th>SSD+Leap Prefetcher</th>
<th>SSD+Read-Ahead</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDD w/o Prefetching</td>
<td>412.32</td>
<td>297.42</td>
<td>204.56</td>
</tr>
<tr>
<td>SSD w/o Prefetching</td>
<td>297.42</td>
<td>254.56</td>
<td>264.93</td>
</tr>
<tr>
<td>HDD+Leap Prefetcher</td>
<td>412.32</td>
<td>297.42</td>
<td>204.56</td>
</tr>
<tr>
<td>HDD+Read-Ahead</td>
<td>297.42</td>
<td>254.56</td>
<td>264.93</td>
</tr>
</tbody>
</table>

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

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

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.](image)

proved timeliness over Read-Ahead (4–52.6×) at the 95\textsuperscript{th} 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 95\textsuperscript{th} percentile. Inclusion of the prefetcher ensures sub-μs 4KB page access latency up to the 85\textsuperscript{th} percentile and improves the 99\textsuperscript{th} 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) disaggregation 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 hits, 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 99\textsuperscript{th} 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 99\textsuperscript{th} 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 99\textsuperscript{th} 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 accesses)
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× 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× and 22.62×, respectively, over the state-of-the-art. This, in turn, leads to application-level performance improvements of 1.27–10.16×. 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

We want to thank the anonymous reviewers, our shepherd, Vincent Liu, and SymbioticLab members for their insightful comments and feedback that helped improve the paper. This work was supported in part by National Science Foundation grants CNS-1845853, CCF-1629397, and CNS-1617773.

References


