# Cori: Dancing to the Right Beat of Periodic Data Movements over Hybrid Memory Systems

Thaleia Dimitra Doudali
*Georgia Institute of Technology*
thdoudali@gatech.edu

Daniel Zahka
*Georgia Institute of Technology*
dzahka3@gatech.edu

Ada Gavrilovska
*Georgia Institute of Technology*
ada@cc.gatech.edu

***Abstract*—Emerging hybrid memory systems that comprise technologies such as Intel's Optane DC Persistent Memory, exhibit disparities in the access speeds and capacity ratios of their heterogeneous memory components. This breaks many assumptions and heuristics designed for traditional DRAM-only platforms. High application performance is feasible via dynamic data movement across memory units, which maximizes the capacity use of DRAM while ensuring efficient use of the aggregate system resources. Newly proposed solutions use performance models and machine intelligence to optimize which and how much data to move dynamically. However, the decision of when to move this data is based on empirical selection of time intervals, or left to the applications. Our experimental evaluation shows that failure to properly configure the data movement frequency can lead to 10%-100% performance degradation for a given data movement policy; yet, there is no established methodology on how to properly configure this value for a given workload, platform and policy. We propose Cori, a system-level tuning solution that identifies and extracts the necessary application-level data reuse information, and guides the selection of data movement frequency to deliver gains in application performance and system resource efficiency. Experimental evaluation shows that Cori configures data movement frequencies that provide application performance within 3% of the optimal one, and that it can achieve this up to 5× more quickly than random or brute-force approaches. System-level validation of Cori on a platform with DRAM and Intel's Optane DC PMEM confirms its practicality and tuning efficiency.**

***Index Terms*—Hybrid Memory, Optane Persistent Memory, Tuning, Data Movement Frequency, Page Scheduler, Data Tiering**

## I. Introduction

Big data analytics, machine learning workloads and data intensive scientific simulations need massive main memory capacities to accelerate data retrieval times and overall application performance. To satisfy this demand, memory hierarchies have become more complex, incorporating emerging memory technologies and disaggregation techniques in order to offset the skyrocketing cost that DRAM-only systems would impose. For example, the Intel® Optane™ DC Persistent Memory (PMEM) platform introduces a high-density, non volatile memory technology at least 3× *slower* than DRAM [2], [21], but also 2×-3× *cheaper* than DRAM [3]. Server configurations with 6 TBs of PMEM pagkaged together with 375 GB of DRAM, such as the Optane platform used in this work, can significantly boost application performance with proper dynamic data management [34], [35].

There are two primary ways to organize hybrid memory hierarchies. One is a vertical organization (cache mode), where one memory type acts as a cache for the other and is managed by hardware. The other is a horizontal organization (flat mode), where all memories 'lay flat' and are managed by software – the operating system or applications themselves. These correspond to the *Memory* and *App-direct* modes in Intel's Optane DC PMEM platform, and each mode introduces different trade-offs with respect to system resource efficiency and application performance. For instance, recent work has shown that the cache organization improves performance of graph applications [17]. In contrast, the flat organization allows for lower energy cost and higher bandwidth use [34], [35], and a number of hardware and software techniques have recently been proposed to further improve the associated management overheads [5], [19], [23], [24], [32], [36].

Prefetching solutions speculatively bring select data from slower to faster memory, such as PMEM to DRAM, when hybrid memory is organized in cache mode. Recent advances in prefetching include novel access pattern prediction methods that use machine learning [20] and dynamic windowing together with majority voting techniques [31]. In contrast, data tiering solutions dynamically rearrange data allocations across *flat* hybrid memory, such that frequently accessed data resides in fast memory (e.g., DRAM), thus maximizing its use. Recent advances in data tiering incorporate machine learning methods into the data selection and movement process [11].

Data tiering solutions often include a *page scheduler* that monitors data access behavior, and periodically migrates data across hybrid memory tiers. While a significant body of research focuses on optimizing the selection of *which* data to move, there is little insight towards *when* that data should be moved. Focusing on the latter, Table I summarizes the operational frequencies of related data tiering solutions, whose difference in time ranges four orders of magnitude. These values are *empirically* tuned to meet the performance requirements of the specific pool of applications evaluated for their respective systems.

Empirical tuning of page scheduling frequency can miss significant performance improvements by not testing certain frequency ranges in an effort to minimize tuning overhead. For example, a common approach [24], [32] is to experiment with period durations that are an order of magnitude apart, e.g., 0.01 sec, 0.1 sec and 1 sec, so as to identify in only

| Solution | Period Duration | Requests per Period |
|---|---|---|
| Thermostat [5] | 10 sec | 100,000 |
| Nimble [40] | 5 sec | 50,000 |
| Ingens [25] | 2 sec | 20,000 |
| HMA [32] | 1 sec | 10,000 |
| Hetero-OS [23] | 0.1 sec | 1,000 |
| Kleio [11] | 0.01 sec | 100 |
| Unimem [38] | MPI phase | N/A |

TABLE I: Frequency of data monitoring and movement across existing solutions mapped to our simulation-based analogy.

| Application | Kernel | Suite | Domain |
|---|---|---|---|
| Back Prop. | `backprop` | Rodinia | Machine Learning |
| Kmeans | `kmeans` | Rodinia | Machine Learning |
| HotSpot | `hotspot` | Rodinia | Physics Simulation |
| LU Decomp. | `lud` | Rodinia | Linear Algebra |
| Breadth-First | `bfs` | Rodinia | Graph Algorithms |
| B+Tree | `bptree` | Rodinia | Databases |
| Pennant | `pennant` | Coral-2 | Hydrodynamics |
| Quicksilver | `quicksilver` | Coral-2 | Monte-Carlo |
| CP Decomp. | `cpd` | ParTI! | Sparse Tensors |

TABLE II: Application kernels used in experiments.

three trials which one offers the highest DRAM hitrate while maintaining reasonable data movement overhead. On the other hand, exploring all frequency choices leads to impractical tuning overheads. In addition, the periodic solutions in Table I fix their operational frequency at the system-level, so that they do not have to repeat the empirical tuning for every application. However, this can potentially leave a significant amount of unexploited performance for applications with data access behaviors and sizes that the empirical tuning did not consider. Another approach is to completely rely on the application to explicitly control data allocation and movement, via use of specialized pragmas or `malloc`-like APIs. Such modified applications then explicitly control how the underlying system-level solution maintains the necessary state to dynamically manage data tiering across hybrid memory [6], [15], [38], [39].

**Problem Statement.** Impractical tuning overheads and lack of insight force existing data tiering solutions to rely on empirical tuning of their operational frequency, or on application-level modifications suitable for specific execution models and APIs. As a result, for general scenarios where modifying the applications is not appropriate, there can be significant levels of performance that existing data tiering solutions do not realize, due to their empirically-tuned and fixed operational frequency.

**Paper Contributions.** To address this, we propose **Cori**[1] – a system-level solution for tuning the operational periods in page schedulers, that maximizes the effectiveness of the schedulers in terms of application performance and platform efficiency, and achieves that with low tuning overheads. Cori operates in an application and runtime-agnostic manner, and relies on observation-based insights to guide the frequency tuning process to a small number of viable candidates. We demonstrate that Cori is effective, it can provide performance gains across applications with different data access behaviors and for different page scheduling policies, and can be practically integrated into the existing hybrid memory management software stack.

The specific contributions of this paper are the following:

- We demonstrate that current data tiering solutions can experience 10%-100% performance loss due to sub-optimal choice of their operational frequencies (Section III-A).
- We identify a relationship among observable application properties – their data reuse – and the favorable scheduling periods (Section III-C).
- We describe the design of **Cori** and its frequency tuning methodology, for a simulation-based prototype and in real system settings. (Section IV).
- We evaluate Cori, demonstrating its ability to identify operational frequencies which realize performance improvements within only 3% from the ideal frequency selection, on average, across applications and page schedulers. Cori achieves this with 5× fewer number of tuning trials, compared to insight-less tuning approaches (Sections V-A, V-B).
- We validate Cori's insights, effectiveness and practicality on a real hardware testbed with DRAM and Intel's Optane DC PMEM (Section V-C).

[1]The name is inspired by the ancient Greek mythology, where Cori (short for Terpsichore) was the muse of dance and daughter of Mnemosyne, the goddess of memory.

## II. Methodology

**Applications.** Table II summarizes the applications that we selected for experimental evaluation from the Rodinia [7], Coral-2 [1] and ParTI! [27] benchmark suites. The selected benchmarks and mini-apps cover a wide range of application domains and memory access patterns.

### *A. Optane DC PMEM Platform*

We have access to a server with Intel Optane DC Persistent Memory Modules (PMEM), which we configure in *App Direct* mode. The machine contains 375 GB of DRAM and 6 TB of PMEM. We implement a page migration module[2] for Linux kernel version 5.4 that attaches to a target process and periodically selects 4 KB pages to move between DRAM and PMEM. Every period, we identify page accesses using the available OS-level information, as also done in [19], [23]. In more detail, the module determines which pages were accessed by scanning the target's page table entries and recording whether or not each accessed bit was set during that period. All accessed bits are then cleared so that they can be tested again during the next scan. To estimate the page hotness, we calculate the exponential moving average (with a certain smoothing factor) of the page's accessed bit history and compare it with a hotness threshold that classifies a page as hot or cold, as also done in [25]. Then, utilizing the `move_pages()` function from the kernel's NUMA-based migration API, we asynchronously move hot pages to DRAM and cold pages to PMEM. The kernel module dynamically adjusts the page migration cutoff,

[2]https://github.com/GTkernel/x86-Linux-Page-Scheduler.git

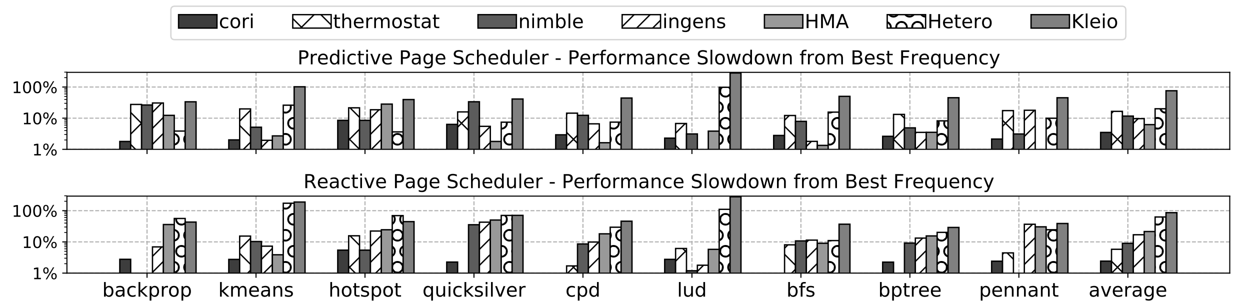


Fig. 1: Performance comparison of a predictive and reactive page scheduler across operational frequencies of existing solutions and the proposed solution Cori, given a simulated hybrid memory system with DRAM and PMEM.

dividing the process memory footprint across DRAM and PMEM at a certain capacity ratio, such as 20% DRAM and 80% PMEM, across all experiments in this paper.

## B. Simulation

**Memory Access Trace Collection.** We use Intel's Pin [4] dynamic binary instrumentation tool to capture the memory address of the last level cache misses out of a simulated three level data cache hierarchy. In order to allow for reasonable trace sizes and analysis times we simulate a cache hierarchy of smaller but proportional capacity ratio to the Intel Optane DC PMEM platform. Then we fix the application data inputs such that we observe similar last level cache miss rate to application execution in the native PMEM platform.

**Hybrid Memory System.** We develop a Python-based simulation environment[3] that allows fast trace-based analysis similar to [11], [32]. In particular, we assume a flat organization of fast (e.g., DRAM) and slow (e.g., PMEM) memory, similar to the App Direct mode configuration of the Intel Optane platform. Following the observed PMEM access speeds [21] we set a 1:3 latency and 1:0.37 bandwidth ratio between the fast and slow memory. We assume that the overall capacity of the memory system is equal to an application's memory footprint, split into 20% DRAM and 80% PMEM across all experiments. Since we are not using cycle-accurate simulation, we assume that a period is the time duration when a fixed number of memory requests are issued, e.g., 1,000 requests per period. To estimate the runtime we aggregate the access latency of the memory requests for their coresponding memory allocation across periods. In addition, we account for any limited bandwidth availability, by injecting appropriate delays given the number of memory requests serviced over a window of time. Finally, we add constant delays for every page migration and start of a period to account for the overhead of the page scheduler itself, using the proposed values in [24], [32].

**Page Scheduler.** We extend the Python-based simulation with a page scheduler that periodically aggregates per page access counts from the collected access trace and migrates pages between fast and slow memory. The initial page allocation is done in an interleaved manner across memories, which is typical for NUMA systems. Every period the page scheduler identifies the pages that are frequently accessed (hot) and moves to fast memory any hot pages that reside in slow memory, replacing any least recently used (LRU) pages. The number of page migrations per period is capped by the available fast memory capacity, since hot and LRU pages are swapped across hybrid memory. These page swaps happen asynchronously, assuming DMA support, and sequentially in order of (hot, LRU) page pairs.

We refer to this type of page scheduler, that makes a selection of page migrations using access history, as a **reactive** page scheduler, since it 'reacts' to the changes in the memory access pattern, as also done in [5], [19], [23], [25], [32], [40]. We also simulate a **predictive** page scheduler, that predicts memory access patterns, thus makes a more sophisticated page migration selection or even has a-priori knowledge of the access pattern, described as the oracular baseline in [11], [32]. The reactive page scheduler is configured to act upon a single period of past access history, and similarly the predictive page scheduler to make an access pattern prediction for the upcoming period.

**Comparison with existing solutions** aims to capture the application performance impact caused only by the selection of *when* to move data, not which and how much data to move. For this purpose we assume the aforementioned page scheduling implementations and compare their behavior with data movement frequencies of existing solutions, as summarized in Table I. Since these proposed values vary across orders of magnitude, we create corresponding period durations that map to our previously described runtime simulation.

## III. Motivation

## A. Performance Gap

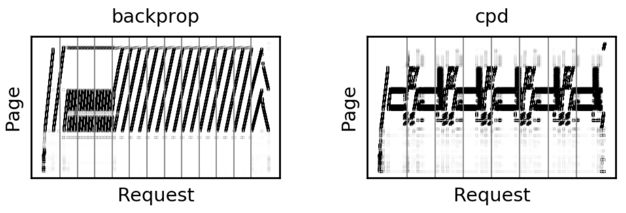
The big disparity in the proposed page scheduling frequencies across related works, summarized in Table I, hints that they are empirically tuned to work best for their given page scheduling implementations and evaluated applications. For this reason, we capture the application performance gap created by using these proposed frequencies as opposed to an optimal frequency across a wide range of data access patterns. Figure 1 captures

[3]https://github.com/GTkernel/cori-sim.git

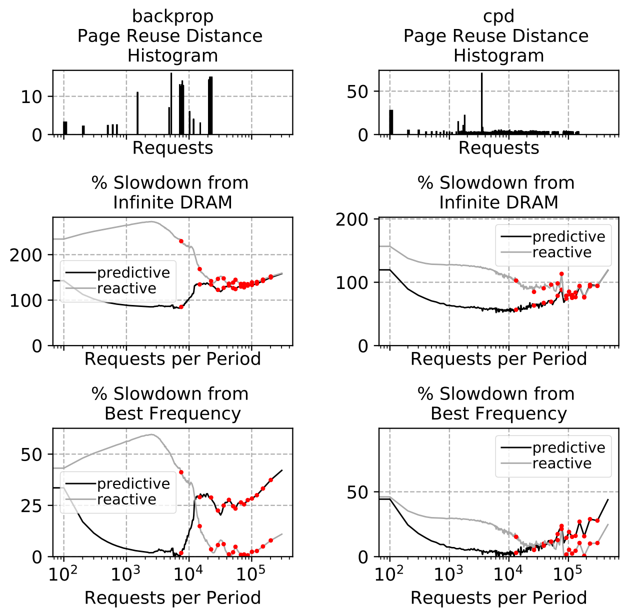
application runtime slowdown from the case of an optimal frequency that provides best performance for each application. The optimal (best) frequency is derived via exhaustive experimental evaluation of a wide range of possible frequencies for every application, as described later in Section III-C. The performance of our proposed solution Cori is also included in the figure, but will be further analyzed in Section V-A.

The proposed frequencies create a 10%-100% performance slowdown compared to the performance achievable with a best-case frequency, on average, across applications and page schedulers. This makes a case for the need for a more robust tuning approach than the empirical one. Taking a closer look, we observe that no single frequency works best across applications and page schedulers. On average, predictive page schedulers experience the lowest performance slowdown when operating at periods of 1 second, as configured in `HMA` [32]. In contrast, reactive page schedulers benefit from periods that are an order of magnitude longer, that is 10 seconds, as configured in `thermostat` [5]. Additionally, the frequency that works best on average for a certain page scheduler may not provide best performance across *all* applications. For example, the lowest slowdown for a reactive page scheduler provided by `thermostat` is not the best choice for `pennant`, `lud`, `hotspot` and `kmeans`. In particular, it incurs up to 40% slowdown from the respective best *proposed* frequency, that is additional to the slowdown from the best frequency itself.

**Takeaways.** This initial experiment validates our concern that frequencies proposed by existing solutions leave a significant performance gap when applied across different applications and page scheduler designs. No single proposed value works best across all applications and page schedulers, and we observe this gap to be in the range of 10%-100%. For large-scale high performance computing and datacenter systems, even small percentage difference can have major implications on cost [12]. Therefore, it is important to close this gap by tuning the operational memory management frequency such that applications maximize their performance across execution platforms and page scheduling policies.



(a) Representative memory access traces. The vertical lines correspond to the fixed period boundaries that provide best performance, as selected by Cori.



(b) Histogram of page reuse distance and its relationship with application performance across period durations, for a predictive and reactive page scheduler over a simulated platform with DRAM and PMEM. The red dots correspond to the performance of the candidate frequencies generated by Cori.

Fig. 2: Performance relationship of the page reuse distance with the period duration (requests per period) of the page scheduler.

## *B. Tuning Overheads*

Existing solutions choose to empirically tune their page scheduling frequency and fix it across applications, to avoid the non-trivial tuning overheads of fine-grained frequency exploration. Stated more formally, an empirical tuning approach has constant time complexity, since it chooses upon a constant set of frequencies. However, the choice of the frequencies themselves is critical, since an insight-less selection can lead to the aforementioned performance gap.

An exhaustive tuning approach has linear time complexity, because the number of possible frequencies grows linearly with the application runtime. For example, the possible period durations for an application that generates $M$ memory requests in total, are the windows of any length between $[1, \frac{M}{2}]$ memory accesses, assuming that a page scheduler should run for at least two periods of $\frac{M}{2}$ requests each. Similarly, if we consider the time domain instead of the memory request domain, the number of possible period durations is such that is splits the application runtime at multiples of a timestep, where a timestamp could be related to the Linux scheduling time slice, for instance.

**The need for some insight.** The long runtime of applications that require massive hybrid memory systems makes an exhaustive tuning approach completely impractical. Instead, we need a more insightful tuning method that can drastically reduce these overheads, and also eliminate the performance gap caused by a poor choice of operational frequency made by empirical selection approaches.

## *C. 'Don't Break the Data Reuse' Insight*

To identify insights that can guide the tuning of the frequency, we first perform exhaustive tuning to determine the best frequency for a given scenario. We select applications with a wide range of data access behavior. For brevity Figure 2a shows a visual representation of the memory access patterns only for `backprop` and `cpd`; we present similar analyses with additional applications in an extended version of this work [13]. We can distinguish the strided array traversals of `backprop` vs. the distinctly shaped sparse tensor traversals of `cpd`.

The top graphs in Figure 2b depict information on data reuse. In the context of these analyses, we use page reuse distance as a measure for page reuse, where the page reuse distance is the number of memory accesses that are issued to other pages, between two consecutive accesses to a particular page. There is a clear connection between the page reuse distances and the access patterns in Figure 2a. For example, for `backprop` the reuse distance of 20,000 requests maps to the gap between the large access strides, and it appears 15 times since there are 16 strides.

**Relation of Performance and Data Reuse.** The bottom graphs in Figure 2b capture an exhaustive exploration of the application runtime slowdown from the case of infinite DRAM capacity and from the case of optimal frequency selection, across all possible period durations for predictive and reactive page schedulers. The x-axis is aligned with the histogram (top graph) and aims to capture the relation between the page reuse distances and page scheduling period durations.

We observe that predictive page schedulers, which make a better selection of which pages to move, provide best application performance for much shorter periods than reactive ones. However, irrespective of the page scheduler's effectiveness, very short periods create a significant aggregate data monitoring and movement overhead, as also shown in Figure 1. In addition, arbitrarily long periods do not allow the page scheduler to react promptly to changes in the access pattern behavior, and thus create insufficient data movement to dynamically improve the data tiering.

Moreover, the effectiveness of reactive page schedulers suffers at periods whose length is shorter than the page reuse distances which appear frequently, incurring an average of 50% additional performance slowdown compared to predictive schedulers. For example, this is the case for `backprop` when periods are shorter than 20,000 requests per period, which is the page reuse distance of its strided access pattern. The scheduler's effectiveness drops because its reactive design identifies as hot pages the ones that correspond to a certain part of the access stride, then moves them to the limited DRAM capacity, but they will not be accessed in the next period, when the rest of the pages of the stride will be accessed. Such reactive page scheduling approaches are more effective when they operate over larger windows of access history, enabled either by longer periods or longer history of shorter periods. Regardless, the time window of access history should be large enough to not 'break' the data reuse.

**Lessons learned.** This extensive application performance characterization shows a clear relationship among the data reuse times and the page scheduling period durations which provide best performance. Reactive page schedulers benefit from periods that don't break the data reuse, to make better page migration decisions. Both reactive and predictive schedulers should avoid very short periods that reveal the data monitoring and movement costs, as well as arbitrarily long periods that do not allow a prompt response to changes in the data access pattern and create insufficient aggregate data movement.

## IV. Solution

**Design Goals.** The objectives of our proposed frequency tuning solution are as follows:

**G1** *Bridge the performance gap* left by existing solutions that do not properly tune their page scheduling frequency.

**G2** *Drastically reduce the number of tuning trials* needed to find the frequency that enables desired performance.

**G3** *Build a generic tuning approach* that works across applications and page schedulers.

**G4** *Enable practical system-level integration* using readily available information on application data access behavior, without explicit code-level modifications or specific APIs.

To address these goals, we propose **Cori**, a method for tuning the data movement frequency in hybrid memory systems. Cori gleans data-movement requirements based on application-specific data reuse trends to guide the frequency tuning process, and select a frequency which delivers performance gains or increases in data movement efficiency (**G1**) with a small number of tuning trials (**G2**). Cori extracts the necessary information from execution profiles, and does not require any changes to applications or the memory management stack (**G3**). Experimental results from a real testbed with DRAM and Intel Optane PMEM validate the simulation-bases evaluation of Cori, and demonstrate the feasibility of its system-level integration (**G4**).

**Cori Overview.** Figure 3 illustrates the system design of Cori and its interactions with the hybrid memory page scheduler, summarized as follows:

1. The Reuse Collector executes a single profile run of the application to collect information on data reuse.
2. The Frequency Generator analyzes the data reuse profile and generates a range of proposed data movement frequencies. To achieve this, it first calculates the dominant reuse period as a weighted average of the observed reuses (2a). Then, it generates a range of candidate frequencies at time intervals that are multiples of the dominant reuse period (2b), and outputs the frequencies to the Tuner in decreasing order, from higher to lower frequencies, thus shorter to longer periods.
3. The Tuner makes a number of tuning trials with the candidate frequencies in the proposed order. It configures the page scheduler to operate at each of the recommended frequencies (3a). It then observes the application runtime and resource use and determines whether the application performance has reached best or desired levels (3b). If not,

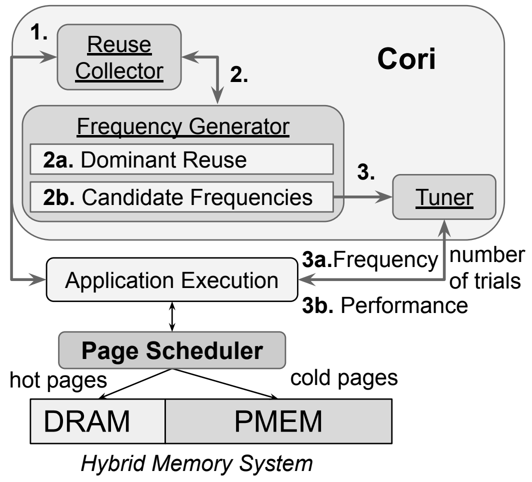


Fig. 3: System components of Cori and its integration with the hybrid memory software stack.

the Tuner moves on to the next frequency in order, going back to step 3a.

Next, we describe in more detail these steps and system components.

## A. Reuse Collector

The goal of the Reuse Collector component is to generate a histogram of data reuse similar to the ones shown in Section III-C. In the context of the simulation-based analysis we collect memory access traces and have access to detailed information on data reuse in terms of page reuse distances at the granularity of each individual memory access. This cannot be generally achieved for arbitrary applications, therefore, we propose a practical system-level alternative to collect similar information on data reuse.

**Loop Durations.** We make the intuitive observation that data reuse appears mostly within loop operations during application execution. Therefore, information on the time duration of loops can be a practical estimation to page reuse distance in the time domain. Figure 5a depicts the time duration of loops across applications including `backprop`. We observe a similar histogram shape to the ones generated via the memory access traces for the page reuse distances in Figure 2b: `backprop` has distinct loop durations that repeat around 15 times, which corresponds to the 16 data access strides depicted in Figure 2a. We validate that the loop duration histograms of the remaining applications match what we observed via the memory access trace collection.

**Collection of Loop Durations.** In the context of validating Cori on a native testbed in Section V-C, we instrumented the applications source code and individually timed the duration of the primary for loops. In principle, however, such instrumentation can easily be performed using compiler-level [10], [18] or binary instrumentation techniques [16], [33]. In the current paper, we do not present a complete Cori tool which integrates such techniques, rather we focus on establishing the methodology that forms the basis of such a tool, and demonstrate via manual instrumentation that the methodology is effective. We verify that we can obtain accurate loop timings using a LLVM compiler pass, similar to what has been used as part of the Beacons compiler framework [10], which automatically generates the instrumented binary without any application source code modifications.

## B. Frequency Generator

**Dominant Reuse.** The Frequency Generator analyzes the data reuse histogram provided by the Reuse Collector, in order to identify the one that best represents the range of captured reuses. We refer to this as the *dominant reuse*. Dominant reuse (DR) is computed as a weighted average of the observed data reuses ($N$ different reuses) in the histogram, as summarized in Equation 1. The weights are the number of appearances $repeat_i$ of a reuse distance $reuse_i$ in the corresponding histogram. This will shift the average towards the data reuse distances that repeat more times. Additionally, we introduce an extra weight $(N-i)$ that favors shorter reuse distances, because this will allow us to generate a more calibrated selection of candidate frequencies, that works irrespective of the page scheduler's effectiveness, as we show in Section V.

$$DR = \frac{\sum_{i=1}^{N}(N-i) \times repeat_i \times reuse_i}{\sum_{i=1}^{N}(N-i) \times repeat_i} \tag{1}$$

$$CandidatePeriods = [DR,\ 2 \times DR,\ ...,\ \frac{Runtime}{2}] \tag{2}$$

**Output Candidate Frequencies.** Based on DR, the Frequency Generator creates a sequence of candidate data movement periods at time intervals that are multiples of DR, as shown in Equation 2. The last possible candidate in the sequence is the one that splits in half the overall application runtime that the Reuse Collector has previously observed. The candidate frequencies are derived by simply inverting the values of the candidate periods. Figure 2b includes a visual representation of the candidate periods as red dots. Finally, the Frequency Generator outputs to the Tuner the candidate frequencies in the specified order from shorter to longer periods, thus higher to lower data movement frequencies. This priority ordering, together with the dominant reuse calculation, is essential to Cori's success, compared to other possible solutions, as shown in Section V-B.

## C. Tuner

The Tuner uses the sequence of candidate frequencies to perform the actual tuning procedure. The Tuner starts its initial trial with the first frequency in order, sets it as the operational page scheduling frequency and executes the application over the hybrid memory. If performance is within desired levels or the best one observed (after the first trial), the Tuner chooses to stop or continue the tuning process. When the Tuner finds the frequency that provides best performance after a number of trials, the selected frequency is kept for any subsequent execution of the particular application on the given combination of platform configuration and page scheduler.

## D. Discussion

Cori currently improves upon tuning approaches, such as the empirical ones, by observing best performance across a number of tuning trials of actual application execution. The decision of after how many trials the tuning stops is flexible. There can be a fixed number of trials or tuning can stop after performance reaches desired levels or shows no significant variation from the last trial. Such an execution-based tuning methodology may be impractical for long running applications, such as training machine learning models and scientific simulations.

However, one can envision future online solutions that build on Cori's methodology and rely on observations made about applications during the initial execution intervals or periodically. Importantly, Cori only requires the collection of data reuse information, that can be made readily available using compiler-assisted instrumentation, laying the grounds towards such a practical online frequency tuning solution. Cori can be extended with system-level performance metrics and combined with online access pattern detection solutions used in prefetching [20], [31], or machine intelligent page schedulers [11], so as to adapt the page migration frequency to dynamic changes in data reuse and access patterns.

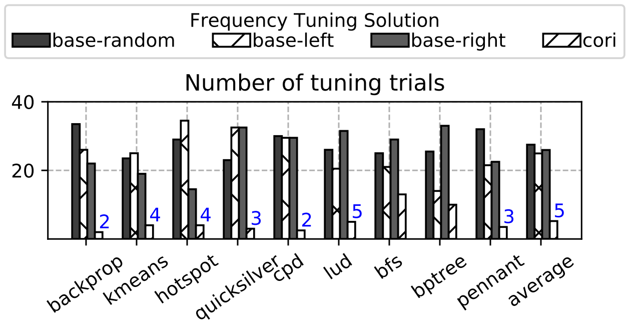
Finally, the recommendations made by Cori depend on the calculation of the dominant reuse, and are therefore sensitive to the granularity at which the data reuse information is collected and aggregated. The evaluations presented in this paper base the calculation on reuse information captured at granularity of 1000s of data accesses (in the simulation framework) and of each loop (on the real hardware testbed). This instrumentation granularity can be dynamically adjusted to trade among the tuning overheads vs. the quality of the recommendations.

## V. Evaluation

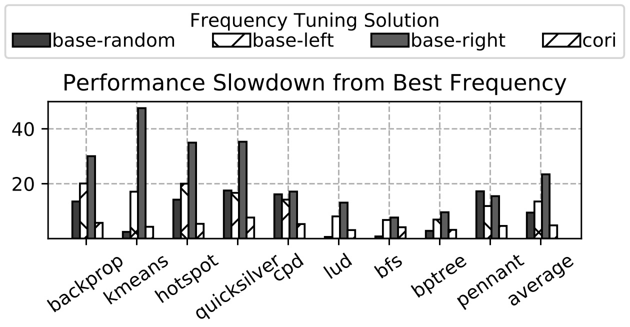
The goal of the evaluation is to demonstrate how Cori realizes its design goals. First, we highlight the benefit of using Cori with respect to application performance improvements. Second, we evaluate the tuning overheads of using Cori. Finally, we validate the effectiveness and practicality of Cori on the native Intel Optane platform.

### A. Benefit of Using Cori

Figure 1 includes the application performance when the page scheduling frequency is tuned with Cori, compared to the frequencies proposed by existing solutions. Cori achieves on average a 3% slowdown via its sophisticated frequency selection, compared to when using the 'best' possible frequency for each of the applications. The 'best' frequency corresponds to the one that maximizes application performance, and it is determined via extensive experimental evaluation of a wide range of frequencies, as shown in Figure 2b. In comparison, the frequencies used by other techniques result in an average 10%-100% slowdown from the ideal case. For cases where Cori does not provide the best application performance, as in the case of `quicksilver` with a predictive page scheduler, the performance with Cori is less than 3% away from the best



(a) Number of tuning trials to find best performance, on average, across page schedulers. Cori (blue text) requires the minimum number of trials on average across applications *and* page schedulers.



(b) Performance slowdown from best frequency for Cori's number of trials, on average, across page schedulers. Cori is the only solution that provides lowest slowdown consistently across applications *and* page schedulers.

Fig. 4: Comparison of Cori with other baseline frequency tuning solutions for a simulated hybrid memory system.

observed one. As discussed in Section III-A, no other set of frequencies proposed by existing solutions provides as good performance across applications *and* page schedulers, as Cori.

*Cori meets the* **G1** *design goal by bridging the performance gap left by existing solutions and achieves only a 3% average slowdown from an optimal frequency selection across applications and page schedulers.*

### B. Overhead of Using Cori

We evaluate the overheads of using Cori by comparing the number of tuning trials required by Cori to find the best frequency vs. what is required to find that value using other tuning methods. We also evaluate whether Cori's overheads are justified, by comparing how close Cori is to the performance of a system which operates at the best possible frequency for each of the applications, vs. how close would the other methods be if they use the same number of trials as Cori.

Given the lack of a non-empirical tuning approach, we construct a **baseline**, which like Cori, operates at the system-level, but is blind to any insights it might have regarding application requirements. This baseline explores the problem space of all possible frequencies by using a simple step function, with candidate periodic time intervals that differ by a time step duration of $\tau step$, as summarized in Equation 3.

The corresponding frequencies are derived by inverting the periodic time intervals.

$$Base\,Candidates = [\tau step,\ 2 \times \tau step,\ ...,\ \frac{Runtime}{2}] \quad (3)$$

Next, we vary the **priority ordering** of the generated candidate frequencies. First, the `base-left` baseline starts from low frequencies (large periods) and moves to the left towards higher frequencies (short periods) in the sequence described in Equation 3. The `base-right` baseline starts from high frequencies and moves towards the right to lower ones, similar to Cori. Third, we also assume a `base-random` approach that randomly explores values in the sequence.

Figure 4a shows the number of tuning trials required to find the frequency that maximizes application performance, on average, across predictive and reactive page schedulers. We include scheduler-specific results in the extended version of this work [13]. We observe that all baseline approaches take a significant number of trials, on average, to find a frequency that maximizes application performance. This is due to their insight-less selection of the exploration time step between tuning trials and the given priority ordering. Even though `base-random` is independent of such a priority ordering, its unpredictable frequency selection results in worst-case average tuning overheads. In contrast, the guided frequency selection performed by Cori, allows it to reduce the number of trials by **5**×, from 25 on average across baselines down to only 5 trials, at the average case.

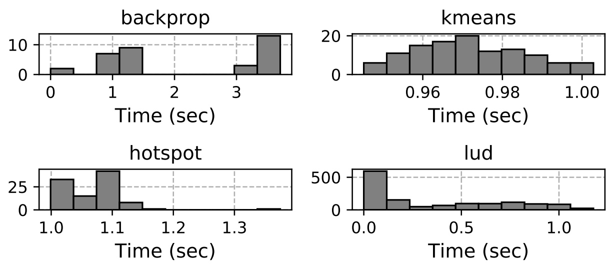
*Cori meets the* **G2** *design goal by reducing by 5× the number of tuning trials needed to reach an average of only 3% performance slowdown, compared to baselines that ignore insights about application data access behaviors.*

Figure 4b shows the performance that the baselines provide when executing for the same number of tuning trials that Cori requires to find best performance. The values are averaged across the page schedulers. On average, the baselines incur higher performance slowdown because they require significantly more trials to reach best performance, as shown in Figure 4a. Within the execution overhead of Cori, only the `base-random` approach seems to be able to still choose frequencies that provide good performance, but only for some of the applications; for others (e.g., `quicksilver` and `pennant`), `base-random` is less effective even compared to some of the other baselines.

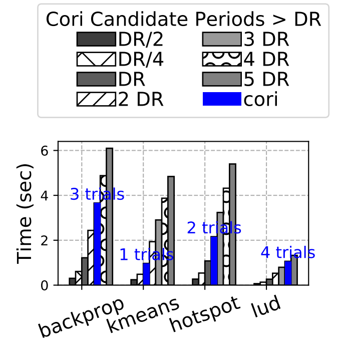
*Cori meets the* **G3** *design goal since it provides maximum performance improvements for minimum number of tuning trials across applications and page schedulers.*

## C. Optane PMEM Validation

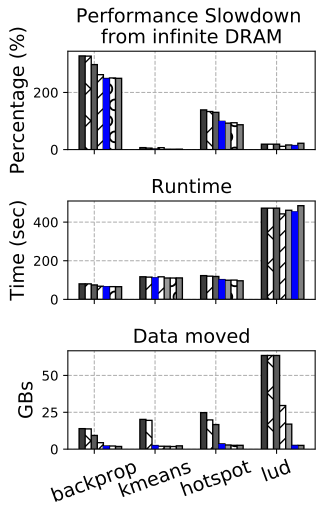
We validate the simulation-based observations about Cori by providing results from a native hybrid memory platform. These experiments also demonstrate the feasibility of using Cori as a practical system-level solution for frequency tuning. The experiments are conduced on an Intel Optane platform, a reactive page scheduler kernel module that operates over a window of past access history, both as described in Section II-A. Then,



(a) **Cori Step 1:** Collect loop time durations.



(b) **Cori Step 2:** Calculate the dominant reuse DR and generate candidate period lengths at multiples of the DR. **Final Choice:** Select the first period length (blue bar) after x trials, that brings lowest runtime *and* migrations.



(c) **Cori Step 3:** Tuning trials of application performance.

Fig. 5: System-level validation of Cori for a reactive page scheduler that executes on a native Optane PMEM platform.

we go through the steps of Cori as summarized in Figure 3 and report our findings in Figure 5.

**Recreating Cori's steps.** First, we gather information on data reuse. More specifically, we collect the time duration of the loops across applications, as shown in Figure 5a, using the suggested approach in Section IV. Second, we calculate the dominant reuse as described in Equation 1 and generate the candidate period durations at multiples of the dominant reuse, as shown in Figure 5b. While for `backprop, kmeans, hotspot` the dominant reuse is around 1 sec, for `lud` it is much less, given the corresponding loop duration histogram. We also include period durations that are less than the dominant reuse, to validate whether performance indeed is not best for such periods that Cori does not include in its sequence of candidates. Third, we replicate Cori's tuning process by executing the applications for the selected period durations in increasing order and observe the runtime, its slowdown from the ideal case of infinite DRAM and data moved, as shown in Figure 5c. The final choice according to Cori is the *first* period duration in the experimentation order that significantly reduces the application runtime and aggregate data movements. Figure 5b indicates in blue the final period choice and the number of tuning trials it required.

**Validation observations.** First, we observe that period lengths that are shorter than the dominant reuse (DR/4, DR/2), create tens of GBs of more data moved, consistently across all applications. This confirms the insight presented in Section III-C that the operational period should not be shorter than the data reuse pattern. Also, it validates Cori's effectiveness in calculating the dominant reuse and choosing it as the initial point of tuning. The performance with much larger periods is not included in Figure 5c, since it can be substantially worse, such as 50% of runtime slowdown for `lud` at 5 second periods, and Cori's tuning ends at much shorter periods.

Second, regarding application performance and system resource efficiency, Cori selects the period duration that reduces to their lowest levels both the data moved and the runtime slowdown from the case of infinite DRAM capacity, across all applications. For some applications these levels of runtime slowdown are less significant than others. For applications like `kmeans` and `lud`, very short periods that force the reactive page scheduler to create a burst of asynchronous data movements, are not enough to stress the Optane's bandwidth and proportionally reflect on their runtime. Regardless, Cori identifies the page scheduling frequency that enables the best performance levels allowed by the available DRAM capacity and minimizes data movement overheads. Additionally, the levels of runtime slowdown observed in this experiment, are very similar to the ones captured in our simulation (Figure 2b), validating its correctness.

Finally, the selected periods themselves are different across applications and range between 1-3 seconds. Even though this doesn't seem as a substantial difference, empirical approaches may have ignored values in such proximity, however, for `backprop` the runtime slowdown reduces by 50% when going from 1 second to 3 second periods, and for `hotspot` by 30% when switching from 1 second to 2 second periods. This validates the benefit from using Cori toward realizing significant application performance improvements, within only 2-3 average tuning trials, minimizing the tuning overheads.

*Cori meets the* **G4** *design goal by allowing for a practical integration with existing hybrid memory system-level managers, and can be realized without modification to applications and system-level components. Validation of Cori on the Intel Optane PMEM platform, confirms the simulation-based motivational arguments and insights, and highlights the benefit of using Cori and its low tuning overhead.*

## VI. Related Work

**Data Tiering across Hybrid Memory.** There is a wide range of data tiering solutions for hybrid memory systems configured as a flat memory address space, that operate across levels of the memory management stack. First, at the application-level there are solutions that propose custom data allocation APIs to improve initial and dynamic data placement [6], [15]. Second, at the runtime-level solutions instrument MPI communication phases and task-based parallel execution to initiate and synchronize the data movement [38], [39]. Moving to the operating system-level (or hypervisor-level) there are solutions which perform periodic data movements using the current NUMA-based page migration support [5], [19], [23], [40], [40], or appropriately extend NUMA-based data balancing policies [14]. Finally, hardware-assisted solutions aim to reduce the data monitoring and movement software overheads [29], [32], [36]. Our work targets operating system-level periodic data tiering solutions and tunes their operational frequency.

**Tuning of System Parameters.** The opportunities for improving performance and efficiency via careful tuning of system-level parameters have been established and demonstrated across different contexts, ranging from operating system-level configuration parameters [22], voltage-frequency balancing for power management [9], CPU scheduling [37], and database index tuning [8]. Such traditional observation-driven tuning techniques are being replaced by reinforcement learning [26], [28], and more generally, learning-augmented solutions are being developed across the systems software stack, and can be unified using machine intelligence frameworks such as AutoSys [30]. Cori uses a traditional observation-driven tuning approach and targets optimizations of the hybrid memory page scheduling frequency, however, the insight from this work can be incorporated in future machine learning-based approaches.

## VII. Conclusion

This work presents Cori, a system-level solution for tuning the operational frequency of data tiering solutions that periodically move data across flat hybrid memory components. Cori synthesizes insights on data reuse information to better guide the process of selecting frequency candidates, reducing by, on average, $5\times$ the number of tuning trials from an insight-less exploration. This way, Cori delivers performance improvements within 3% from the case of optimally chosen frequency, completely eliminating the 10%-100% performance gap created by using operational frequencies adopted across recent related works. Cori is robust, and provides benefits across application data access patterns and page migration policies. Importantly, we validate that Cori's approach is effective in the context of a real system, and that it can be integrated as part of a system-level tuning solution. This is feasible via the use of information that can be automatically extracted through compiler-based methods, without requiring any code-level modifications of applications or across the memory management software stack; our next steps will enhance the system with such capabilities.

## Acknowledgment

We thank the anonymous reviewers on their constructive feedback. This work is partially supported by NSF awards SPX-1822972 and CNS-2016701, a Facebook research grant, and via access to an Optane PMEM hardware testbed provided by Intel.

## References

[1] CORAL-2 Benchmarks. https://asc.llnl.gov/coral-2-benchmarks/, 2020.
[2] Intel® Optane™ DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2020.

[3] MemVerge - More Memory. Less Cost. https://www.memverge.com/more-memory-less-cost/, 2020.

[4] Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html, 2020.

[5] N. Agarwal and T. F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. ASPLOS '17, pages 631–644, New York, NY, USA, 2017. Association for Computing Machinery.

[6] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, and S. D. Hammond. memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies. 3 2015.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[8] D. Dash, N. Polyzotis, and A. Ailamaki. Cophy: A scalable, portable, and interactive index advisor for large workloads. *Proc. VLDB Endow.*, 4(6):362–372, Mar. 2011.

[9] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, page 31–40, New York, NY, USA, 2011. Association for Computing Machinery.

[10] V. Deodhar, H. Parikh, A. Gavrilovska, and S. Pande. Compiler Assisted Load Balancing on Large Clusters. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[11] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska. Kleio: A hybrid memory page scheduler with machine intelligence. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, pages 37–48, New York, NY, USA, 2019. ACM.

[12] T. D. Doudali and A. Gavrilovska. Mnemo: Boosting memory cost efficiency in hybrid memory systems. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 412–421, 2019.

[13] T. D. Doudali, D. Zahka, and A. Gavrilovska. Tuning the frequency of periodic data movements over hybrid memory systems. https://arxiv.org/abs/2101.07200, 2021.

[14] Z. Duan, H. Liu, X. Liao, H. Jin, W. Jiang, and Y. Zhang. Hinuma: Numa-aware data placement and migration in hybrid memory systems. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 367–375, 2019.

[15] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[16] DynInst: Putting the Performance in High Performance Computing. dyninst.org.

[17] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory. *Proc. VLDB Endow.*, 13(8):1304–1318, Apr. 2020.

[18] T. Grosser, A. Groesslinger, and C. Lengauer. Polly - Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4), 2012.

[19] V. Gupta, M. Lee, and K. Schwan. Heterovisor: Exploiting resource heterogeneity to enhance the elasticity of cloud platforms. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 79–92, New York, NY, USA, 2015. Association for Computing Machinery.

[20] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan. Learning memory access patterns. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1919–1928, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[21] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic performance measurements of the intel optane dc persistent memory module, 2019.

[22] S. K. Johnson, B. Hartner, B. Pulavarty, and D. J. Vianney. *Linux Server Performance Tuning*. Prentice Hall PTR, USA, 2005.

[23] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 521–534, New York, NY, USA, 2017. Association for Computing Machinery.

[24] V. R. Kommareddy, S. D. Hammond, C. Hughes, A. Samih, and A. Awad. Page migration support for disaggregated non-volatile memories. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, pages 417–427, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 705–721, USA, 2016. USENIX Association.

[26] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proc. VLDB Endow.*, 12(12):2118–2130, Aug. 2019.

[27] J. Li, Y. Ma, and R. Vuduc. ParTI! : A parallel tensor infrastructure for multicore CPUs and GPUs, Oct 2018. Last updated: Jan 2020.

[28] Y. Li, K. Chang, O. Bel, E. L. Miller, and D. D. E. Long. Capes: Unsupervised storage performance tuning using neural network-based deep reinforcement learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu. Utility-based hybrid memory management. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 152–165, 2017.

[30] C.-J. M. Liang, H. Xue, M. Yang, L. Zhou, L. Zhu, Z. L. Li, Z. Wang, Q. Chen, Q. Zhang, C. Liu, and W. Dai. Autosys: The design and operation of learning-augmented systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 323–336. USENIX Association, July 2020.

[31] H. A. Maruf and M. Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857. USENIX Association, July 2020.

[32] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, volume 00, pages 126–136, Feb. 2015.

[33] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri. LoopProf : Dynamic Techniques for Loop Detection and Profiling. In *Workshop on Binary Instrumentation and Applications (WBIA)*, 2006.

[34] O. Patil, L. Ionkov, J. Lee, F. Mueller, and M. Lang. Performance characterization of a DRAM-NVM hybrid memory architecture for HPC applications using intel optane DC persistent memory modules. In *Proceedings of the International Symposium on Memory Systems, MEMSYS 2019, Washington, DC, USA, September 30 - October 03, 2019*, pages 288–303. ACM, 2019.

[35] I. B. Peng, M. B. Gokhale, and E. W. Green. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 304–315, New York, NY, USA, 2019. Association for Computing Machinery.

[36] A. Prodromou, M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen. Mempod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 433–444, 2017.

[37] M. I. Seltzer and A. Fedorova. Operating system scheduling for chip multithreaded processors. 2006.

[38] K. Wu, Y. Huang, and D. Li. Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 58:1–58:14, New York, NY, USA, 2017. ACM.

[39] K. Wu, J. Ren, and D. Li. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 31:1–31:13, Piscataway, NJ, USA, 2018. IEEE Press.

[40] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee. Nimble page management for tiered memory systems. ASPLOS '19, page 331–345, New York, NY, USA, 2019. Association for Computing Machinery.