INDIGO: Page Migration for Hardware Memory Disaggregation Across a Network

Archit Patke
University of Illinois at
Urbana-Champaign

Christian Pinto
IBM Research

Saurabh Jha
IBM Research

Haoran Qiu
University of Illinois at
Urbana-Champaign

Zbigniew Kalbarczyk
University of Illinois at
Urbana-Champaign

Ravishankar Iyer
University of Illinois at
Urbana-Champaign

Abstract

Hardware memory disaggregation (HMD) is an emerging technology that enables access to remote memory, thereby creating expansive memory pools and reducing memory underutilization in datacenters. However, a significant challenge arises when accessing remote memory over a network: increased contention that can lead to severe application performance degradation. To reduce the performance penalty of using remote memory, the operating system uses *page migration* to promote frequently accessed pages closer to the processor. However, previously proposed page migration mechanisms do not achieve the best performance in HMD systems because of obliviousness to variable page transfer costs that occur due to network contention. To address these limitations, we present INDIGO: a network-aware page migration framework that uses novel page telemetry and a learning-based approach for network adaptation. We implemented INDIGO in the Linux kernel and evaluated it with common cloud and HPC applications on a real disaggregated memory system prototype. Our evaluation shows that INDIGO offers up to 50–70% improvement in application performance compared to other state-of-the-art page migration policies and reduces network traffic up to 2×.

1 Introduction

Background and Motivation. Hardware memory disaggregation (HMD) is an emerging technology that enables cache-line granularity access to remote memory with protocols such as CXL or openCAPI [2, 3, 7], thereby creating expansive memory pools and reducing memory underutilization in datacenters. To cost effectively support the growing requirements of memory-intensive applications [49, 60, 64], memory expansion has gone beyond a single compute node to multiple nodes and a remote memory pool that are connected together with a switched interconnect [5, 9, 52]. As with any switched network, contention from multiple hosts results in slow down of remote memory access as shown in Figure 1. Specifically, we find that multi-node contention in HMD systems results in 3–6× higher performance degradation compared to traditional tiered memory systems [24, 25, 40, 50, 62]. Our discussions with a leading cloud and HPC provider indicate that network contention is a significant bottleneck in the deployment of CXL memory.

Page migration [6] can be reused to address network contention in HMD systems by transferring frequently accessed pages from remote to local memory. However, the associated *page transfer cost*, i.e. the time required to copy pages between remote and local
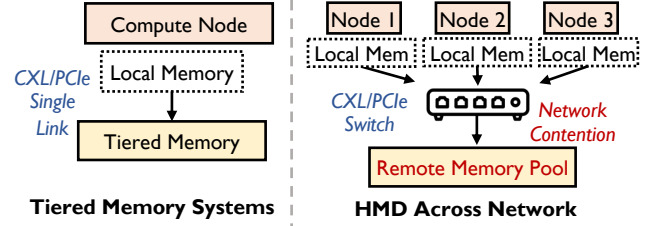

Figure 1: Compared to tiered memory systems, multiple nodes contend for the same remote memory pool leading to network contention in HMD systems.

memory, is higher in HMD systems and increases during network contention as shown in Figure 2 (Left). Previously proposed page migration mechanisms such as [6, 10, 27, 29, 33, 36, 44, 55, 56, 65, 68] that are designed for legacy NUMA systems, or disaggregated systems with point-to-point CPU-memory connections, do not consider such variable page transfer costs resulting in sub-optimal scenarios as described below.
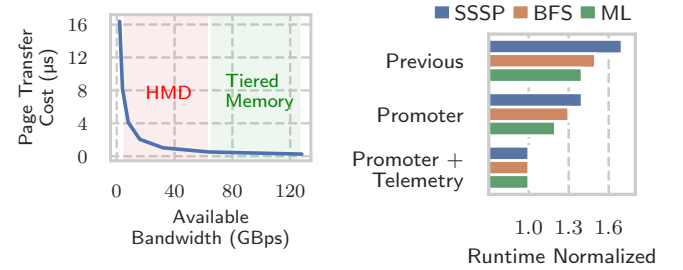

Figure 2: Previously proposed page migration mechanisms do not consider the variable page transfer costs in HMD systems. (Left) Page transfer cost for 4 KB pages in tiered memory and HMD systems under varying network contention. (Right) INDIGO considers the impact of variable page transfer costs with Page Promoter and Page Telemetry resulting in decreased application runtime. Each application was run with local memory allocation = 10% of the working set size and compared with the best runtime between TPP [44], MEMTIS [33] and Nimble [68]

First, previously proposed mechanisms are not effective at transferring pages during network contention. Page migration is only beneficial when the improvement in access locality outweighs the

page transfer cost. During network contention, the cost of transferring pages increases because of reduced available bandwidth, which results in excessive page transfers with the current mechanisms.

Second, previously proposed mechanisms fail to transfer pages optimally when application access patterns are dynamic. Page migration is based on access rates estimated using moving averages [33, 55] or bit vectors [32]. These estimates become highly inaccurate with dynamic patterns. For example, Graph500 applications like BFS and SSSP have estimation errors greater than 1000%, resulting in the transfer of infrequently accessed pages. In HMD systems, this issue is exacerbated due to the high cost of page transfers.

INDIGO Design. To resolve the limitations of previous page migration mechanisms in the context of HMD systems, we introduce INDIGO. INDIGO consists of two major components: *Page Telemetry* and *Page Promoter*.

Page Telemetry improves the accuracy of page access estimation by differentiating between stationary and dynamic application access patterns. Differentiation between the two patterns is enabled by calculating the size of each *interval* (i.e., "burst duration") in which the access pattern does not vary significantly. Smaller burst durations imply that the application memory access pattern is dynamic (and vice versa for larger ones).

Page Promoter adopts a network-aware page migration policy that considers the trade-off between the cost of page transfer and memory access locality benefit (we develop an analytical model to consider this trade-off in Section 2.4) within the constraint of limited local memory allocation. The input to the Page Promoter includes measurements from Page Telemetry, application memory allocation, and network telemetry. The Page Promoter migrates pages if the access rates and burst duration exceeds a threshold value. This threshold value is decided at the application start by a contextual multi-armed bandits agent that is trained using historical application runtime data. We also provide an alternate heuristic-based network adaptive policy if no application training data is available[1].

INDIGO is implemented between Linux kernel and the userspace. Page Telemetry and Page Promoter are both implemented in the Linux kernel, while the contextual multi-armed bandits control is implemented in the userspace. Both components contribute to INDIGO's improved performance as shown in Figure 2 (Right).

Results. We evaluated INDIGO with common cloud applications, including in-memory databases [60], in-memory key-value stores [16], ML inference [64], graph analytics kernels [49], and scientific computational code [20] on a real HMD prototype system: Thymesis-Flow [8]. We compared INDIGO with state-of-the-art page migration techniques and recent research work available in the literature, such as TPP [44], Nimble [68], and MEMTIS [33]. We summarize key results below:

1. *Improved Application Performance.* INDIGO improves application performance by up to 50–70% compared to other state-of-the-art page migration policies across low to high local-to-remote memory allocation ratios. The network adaptive policy also offers an improvement of up to 20–50% in the presence of network contention.

2. *Unlocking Memory Offload.* For applications such as ML inference, INDIGO can match full local memory execution performance with limited local memory allocation, thereby enabling 80–90% offload of the application working set[2] from local to remote memory.

3. *Reduced Network Traffic.* INDIGO reduces network traffic by up to 3× compared to other policies, such as MEMTIS, TPP, and Nimble, while improving the overall application performance. The decreased network traffic leads to reduced contention and improved colocation for multiple applications that share the network.

4. *Minimal Training Costs.* Our evaluation shows that INDIGO, when exclusively trained on a particular application, will perform equally well on other, similar applications without any retraining. For example, there is a <5% drop in application performance when one trains INDIGO on a breadth-first search (BFS) application but tests the trained model on a single source shortest path (SSSP) application (with no retraining/adaptation).

Putting INDIGO in perspective. The problem of deciding the location of memory pages is also present in other OS-based approaches to memory disaggregation that use RDMA along with a swap cache such as InfiniSwap [26], Leap [12], Fastswap [13], and Hermit [53]. However, these RDMA-based systems are fundamentally different from HMD systems, as they emulate disaggregated memory via the kernel; rather than extending the address space. Consequently, *they do not have the option of keeping the accessed page in remote memory*; it has to be brought to local memory. That changes the decision problem, and the analytical model (later presented in Section 2.4) no longer holds. The page-to-transfer decision becomes a prefetching problem as opposed to a migration problem.

2 Background

We discuss the background of HMD systems and page migration, and highlight the drawbacks of state-of-the-art page migration mechanisms when applied to HMD systems.

2.1 Hardware Memory Disaggregation

INDIGO targets hardware memory disaggregation (HMD), which employs cache lines as a basic block for memory access. Similar to tiered memory, the remote memory pool is accessed only in response to a last-level CPU cache miss, without any intervention of the operating system or any other software runtime. Cache lines are transmitted via a high-performance cache coherence protocol such as OpenCAPI [7] or CXL [3]. Additionally, HMD systems have multiple nodes accessing the same remote memory pool as shown in Figure 1. Due to cache line-level access, HMD exhibits the lowest remote memory access latency as it eliminates the software overhead present in other OS-based [11, 13, 19, 21, 26, 30, 34, 38, 59] or custom runtime-based approaches [57, 66, 70] for disaggregation. Regardless of its implementation, accessing of remote memory is generally slower than accessing of local memory due to the higher access latency and lower bandwidth availability that result from the memory network, and performance degradation is expected for most applications. Most HMD solutions [15, 52] reuse the existing OS memory features created specifically for the NUMA architecture and materialize disaggregated memory as a CPU-less remote NUMA node. We adopted the same design choice for INDIGO.

---

[1]The heuristic-based approach is only network adaptive but not adaptive to the dynamicity of application access patterns

[2]The "working set" of an application is the collection of memory pages that are currently owned by the application.

Archit Patke, Christian Pinto, Saurabh Jha, Haoran Qiu, Zbigniew Kalbarczyk, and Ravishankar Iyer

System	Access Estimation	Migration Policy	Network Adaptation	E2E Performance Aware	Page Demotion
autoNUMA [1]	✗	Static Threshold	✗	✗	Recency
autoTiering [29]	✗	Promotion-based	✗	✗	Recency
TPP [44]	✗	Static Threshold	✗	✗	Recency
Nimble [68]	✗	Static Threshold	✗	✗	Recency
Multi-Clock [43]	✗	Static Threshold	✗	✗	Recency + Frequency
HeMem [55]	Moving Average	Static Threshold	✗	✗	Frequency
MEMTIS [33]	Moving Average	Histogram-Based	✗	✗	Recency + Frequency
INDIGO (w/o training)	✗	Network-Based	✓	✗	Recency + Frequency
INDIGO (with training)	Contextual Bandits		✓	✓	Recency + Frequency

Table 1: Comparison of INDIGO with other existing page migration mechanisms in disaggregated and tiered memory systems.

2.2 Network Contention

Memory contention is a well-known phenomenon in computing systems, where multiple application streams or processor cores access memory devices across shared links, sharing the fixed amount of bandwidth available to transfer data to/from memory. Memory contention would be even worse in HMD systems based on a switched network, where a number of nodes access the remote memory pool via one or more switches [9, 51]. For example, the commercial Falcon C5022 from H3 Platform, based on a CXL 2.0 switch [5], enables sharing a single memory device over a PCIe 5.0 x8 link (64GBps) with up-to 8 compute nodes, resulting in an effective bandwidth of 8 GBps per node when they access memory locations in the same device simultaneously.

2.3 Page Migration

A memory page is the logical abstraction of memory at the operating system level. The physical location of each page can be dynamic and can vary between nodes in a NUMA system such that frequently accessed memory is closer to the CPU in which the computation is taking place (local). In this approach, unlike disk-based swapping, a page in remote memory is accessible to the CPU at cache line granularity, albeit with higher latency. To decide the location of each page dynamically, the operating system implements a *page migration* policy that consists of (a) page telemetry, which measures page access patterns; (b) page promotion, which transfers pages from remote to local memory; and (c) page demotion, which transfers pages from local to remote memory. As page access rate may vary over time, page promotion and demotion are always "on" and work together to dynamically decide on whether to place pages in the remote memory pool or on local nodes.

2.4 Network Cost Model for Page Migration

We introduce a network cost model for page migration in HMD systems and leverage it to justify our design choices for INDIGO throughout the rest of this paper.

The model considers a migration (i.e., swap) between two pages **p** (promoted page) and **d** (demoted page) initially located in remote and local memory respectively. Page migration benefits memory access locality (bringing it closer to compute) for page **p** while harming access locality for page **d** and costing time for the page transfer on the network. The advantage of page migration is only realized when the benefit is more than the cost as represented in Equation (1).

$$\underbrace{\int_0^{\Delta T} (\text{access}_p - \text{access}_d)\, dt}_{\text{Access Locality Benefit}} \geq \underbrace{\frac{\text{page size}}{\text{bandwidth} \times \Delta \text{ latency}}}_{\text{Page Transfer Threshold}} + k \quad (1)$$

ΔT is the interval between the current and next swaps of the pages and is determined by the page migration policy. access_p and access_d are the access rates for the pages p and d during the interval ΔT. The network cost of page transfer is represented by $\frac{\text{page size}}{\text{bandwidth} \times \Delta \text{ latency}}$, where Δ latency is the difference between remote and local memory latency. k is the constant associated with the bookkeeping overhead of page migration that includes handling interrupts, remapping of page table entries, flushing of the translation lookaside buffer (TLB), and so on.

2.5 Limitations of Existing Page Migration Mechanisms

Previous work on page migration has adopted different approaches for page telemetry, promotion, and demotion. We summarize these approaches in Table 1.

Page Telemetry. The ideal migration decision is based on an estimate of the difference between the access patterns for the promoted and demoted pages ($\int_0^{\Delta T} \text{access}_p dt$ and $\int_0^{\Delta T} \text{access}_d dt$ in Equation (1)) and whether it exceeds the cost of transferring the page. Other than MEMTIS, no mechanism considers the access rate of the demoted page ($\int_0^{\Delta T} \text{access}_d dt$) during decision-making for page promotion. Instead, they rely solely on an estimate of the access rate of the promoted page. However, such an approach can lead to the promotion of pages even when the access rate of the demoted page is high. This scenario is suboptimal as the benefit of page migration (shown on the left-hand side of Equation (1)) is low or even negative, thereby causing performance loss. Even for estimating the access rate of the promoted page ($\int_0^{\Delta T} \text{access}_p dt$), mechanisms like autoNUMA, autoTiering, TPP, Nimble, and Multi-Clock use the last access measurement without normalization or prediction. Others, such as HeMem and MEMTIS, use variations of exponentially weighted moving averages (EWMA) to smooth out previous measurements. However, EWMA may not be accurate for dynamic workloads. (We characterize this problem in Insight 2, below.) In addition, it is challenging to determine the appropriate decision context (i.e., ΔT) to accurately estimate the relevant time window for the estimated access patterns.
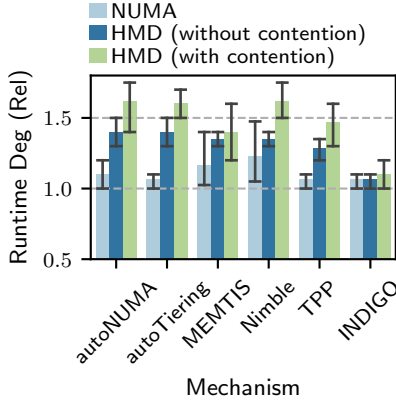
Figure 3: Runtime degradation for differ-ent page migration mechanisms.

Figure 4: Page promotion rates with dif-ferent page migration mechanisms.

Figure 5: Dynamically shifting applica-tion access patterns for a BFS application.

Page Migration. Fundamentally, most previous mechanisms are suboptimal in HMD systems because they do not consider the page transfer cost term in Equation (1). (We characterize this problem in Insight 1, below.) Mechanisms such as autoNUMA, TPP, Nimble, and HeMem have static thresholds for promotion (such as promoting if $\int_0^{\Delta T} \text{access}_p \, dt > $ fixed constant). In particular, autoTiering and MEMTIS adapt thresholds based on different criteria. autoTiering migrates the most recently accessed pages to meet a target promo-tion rate. MEMTIS also considers the access rate of the demoted page and migrates pages such that the local memory allocation is always at capacity. Neither of those mechanisms can easily be modified to support network cost adaptation.

2.6 Core Insights

The above mentioned background knowledge and limitations of prior approaches led to three core insights, that we use to motivate the design of INDIGO.

Insight 1: Existing page migration mechanisms fail to adapt based on changing network conditions. Previous work on page migration did not take into account the network cost of transferring pages: $\frac{\text{page size}}{\text{bandwidth} \times \Delta \text{ latency}}$ (as shown in Equation (1)). This network cost term can be up to 16× higher in HMD systems than in conven-tional NUMA architecture because of the smaller network band-width (especially during multi-tenant contention). Consequently, mechanisms that do not consider network cost perform well for conventional NUMA but make suboptimal decisions when ported to HMD systems, e.g., transferring excessive amounts of pages during periods of network contention.

To characterize that behavior, we ran the Graph 500 BFS appli-cation with 5 different page migration mechanisms in three scenar-ios: (a) a high-bandwidth local NUMA node (NUMA) equivalent to tiered memory, (b) a remote memory pool without additional network contention (HMD without contention), and (c) a remote memory pool under network contention via a memory copy bench-mark (HMD with contention) [3]. Figure 3 shows the relative runtime degradation (normalized by the best run of the best-performing mechanism). Coloured bars show the median value, while error bars

[3]STREAM [14] COPY kernel was used to reduce available bandwidth to 8 GBps to create multi-node network contention.
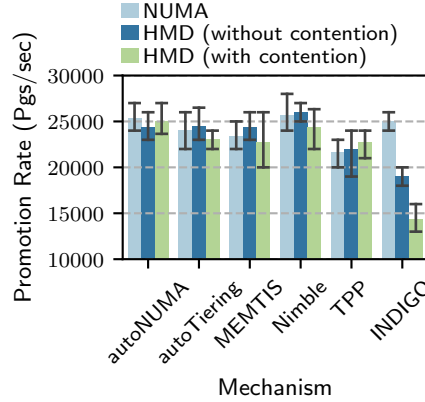
show the range min-max across all runs) of the BFS application under the three scenarios. While most of the page migration mech-anisms performed well on the local NUMA node, their performance suffered in the HMD system: both with and without network con-tention. The performance improvement offered by INDIGO can be explained using Equation (1). Network access causes the cost of mi-grating pages ($\frac{\text{page size}}{\text{bandwidth} \times \Delta \text{ latency}}$) to increase significantly because of the lower available bandwidth (100× lower than for NUMA in our prototype). Consequently, the difference between the access rates of the promoted page and the demoted page ($\text{access}_p - \text{access}_d$) needs to be higher for optimal transfer. Therefore, fewer pages would have access rates that are high enough to justify migration and the overall promotion rate should be lower. Characterization of the promotion rates as shown in Figure 4 confirms this observation. INDIGO is more selective in page promotion, as the promotion rate is about 25–50% lower than that of the other mechanisms.

> Page migration mechanisms must adapt their page promotion se-lectivity based on the network conditions in HMD systems.

Insight 2: Relying solely on moving averages is insufficient for page access telemetry. Page migration mechanisms such as HeMem and MEMTIS use exponentially weighted moving averages (EWMA) to estimate future memory access patterns based on his-torical access records and use them to make the migration decision. EWMA can be very effective at dealing with *stationary* memory access patterns, as minor measurement fluctuations are corrected. However, we find that it can be completely ineffective when dealing with dynamically shifting application access patterns. For instance, consider the access patterns of a segment of the Graph 500 BFS application as shown in Figure 5. As the page access patterns keep shifting dynamically, EWMA is unable to estimate which pages will be frequently accessed causing them to remain in the remote mem-ory pool. For example, EWMA with $\alpha = 0.5$ (used in HeMem) and $\alpha = 0.9$ (used in MEMTIS) leads to p75 error = 170% and p99 error = 60000% when estimating the access rate in the next measurement window.

> There is a need to redesign page telemetry so that it can work well for both *stationary* and *dynamic* application access patterns.
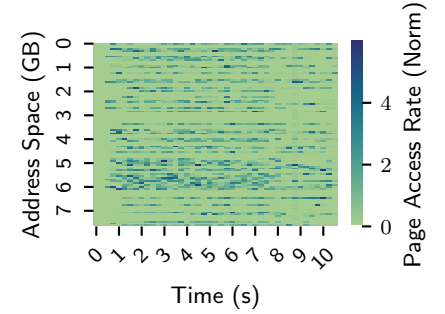
Insight 3: Accurate estimation of page access patterns alone is not sufficient for the page migration decision problem. Equation (1) shows that optimal decision-making depends on estimating $\int_0^{\Delta T} (\text{access}_p - \text{access}_d)\, dt$; that involves the decision context (ΔT) in addition to the page access rates. The optimization problem of selecting the correct ΔT is difficult to solve optimally online. A possible solution is to use algorithmic optimization techniques (such as bipartite matching) paired with a supervised learning model that estimates page access patterns. However, such an optimization requires computation of $\sim O(\text{pages}^3)$ [31], which is prohibitively expensive for large memory allocations (i.e., minutes $\gg$ sub-millisec).

> Classical algorithmic optimization leads to prohibitively high computational costs for making online page migration decisions.

3 INDIGO Design Overview

We derived the design of INDIGO from first principles driven by the insights presented in Section 2.6 and Equation (1). The page migration decision problem relies on estimates of page access rates (access_p and access_d) to migrate pages for the decision context (ΔT). Our design focuses on increasing the estimation accuracy of page access rates via improvements in page telemetry and by optimally solving the decision problem using learning-based techniques.

Design Choice 1: Augmenting page telemetry with contextual (time-based) features for accurate page access estimation. The page migration decision is based on estimating the access rates for both the promoted and demoted pages. Previous mechanisms, such as HeMem and MEMTIS, used exponentially weighted moving averages (EWMA) to estimate these access rates. However, as we show in Insight 2 in Section 2.6, moving averages fail to adapt effectively, especially when application access patterns vary dynamically. On the other hand, moving averages work well with static access patterns when small deviations in access rates can be ignored. Our goal is to design a mechanism that can generalize for both access patterns. To do so, we introduce a new telemetry feature that captures variability in historic access patterns via a "burst duration." The burst duration is the time interval for which the current access pattern remained valid in the past. A smaller burst duration indicates dynamic application access patterns (and vice versa for a larger burst duration). The burst duration adds a time-based dimension to the access measurements necessary for accurate estimation. In Section 4, we show that burst duration is easy to compute within the kernel space and adds minimal overhead.

Design Choice 2: Utilizing a feedback loop to solve the page migration optimization problem. In addition to accurate estimations of future page access rates, the page migration decision problem also requires selection of the decision context (i.e., ΔT). As discussed in Insight 3 in Section 2.6, selecting the appropriate ΔT is difficult via supervised learning because of the high computational overhead ($O(\text{pages}^3)$). As solving the vanilla decision problem is challenging, we simplify it by using additional feedback from application performance. This feedback can be combined with page telemetry to create an end-to-end policy-training loop for the decision optimization problem. The specific optimizer we chose is contextual multi-armed bandits, as it requires less parameter tuning
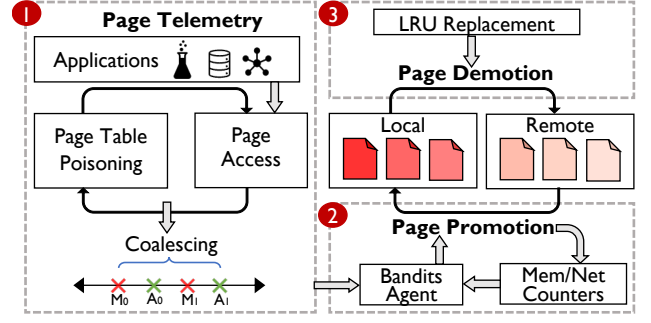

Figure 6: INDIGO Workflow

(compared to out-of-the-box reinforcement learning). We describe details of the optimization loop in Section 5.

Design Choice 3: Enabling dynamic policy adaptation based on application access patterns, network contention, and memory allocation. As described in Design Choice 2, we use an end-to-end feedback loop to model and solve the page migration decision problem. We revisit Equation (1) to decide on the necessary input for this feedback loop. The term access_p is typically measured by page table poisoning and interception of the corresponding page fault. This approach is commonly used in other mechanisms, such as autoNUMA, autoTiering, and TPP, and we use it too. Along with the burst duration, it can be effective for estimating both static and dynamic access patterns (for access_p). The other term access_d is more challenging to estimate as it is not in the critical path of page migration, and page table-based estimates will lag the true access rates. To resolve that problem, we observed that access_d is dependent on the (local) memory allocation for a given application. Pages are ranked by their access rate to decide which ones are eligible for promotion or demotion, and in general those with the lowest access rates would be demoted. Consequently, the larger the local memory available, the larger is the set of pages that are eligible for promotion and the lower is the access rate of the demoted page. Finally, we must take into account the network characteristics that we measure using network hardware performance counters. By taking into account those features, driven by the formulation presented in Equation (1), INDIGO is able to dynamically adapt the page migration policy.

INDIGO Workflow. Figure 6 presents a high-level workflow of INDIGO that uses the design choices discussed here. First, INDIGO Page Telemetry measures page access rates and coalesces similar measurements to estimate burst durations. The burst duration aids in differentiating between static and dynamic access patterns. Next, these measurements are used along with other system metrics, such as memory allocation and network telemetry (collected from memory/network counters, as shown in the figure), to determine the pages that should be migrated. The page migration decision is then taken by a contextual bandit agent trained in a feedback loop using application performance metrics (for e.g. application completion time). If the collection of training data is not feasible, INDIGO also offers a network adaptive policy that adjusts promotion rates based on varying network conditions. Finally, infrequently accessed pages are demoted back to the remote memory pool using the regular Linux LRU replacement mechanism.

Algorithm 1 Coalescing Access Measurements to Estimate Burst Duration (i.e., the Size of the Cluster).

Require: Marking and Access Times $\langle M, A \rangle$
1: **procedure** CLUSTERACCESSRATE($\langle M, A \rangle$)
2:     Clusters $C \leftarrow \varnothing$
3:     Access Rate $F \leftarrow \varnothing$
4:     **for** each $\langle M_i, A_i \rangle$ in $[\langle M, A \rangle]$ **do**
5:         $F_i = 1/(A_i - M_i)$
6:         $sz = \text{size}(C)$
7:         **if** $i > 0$ & $|F_i - F_{i-1}| < \delta 1$ & $M_i - M_{i-1} \leq \delta_2$ **then**
8:             C_{sz-1}.add(i) // Add access rate to curr cluster
9:         **else**
10:             C.append() // Create a new cluster
11:             C_{sz}.add(i) // Add access rate to new cluster
12:         **end if**
13:     **end for**
14:     **return** Clusters C, Access Rate F
15: **end procedure**

---

4 Page Telemetry

In INDIGO, the main goal of Page Telemetry is to enable the estimation of page access patterns to be used in the feedback loop. The Page Telemetry is primarily based on page table poisoning similar to mechanisms such as autoNUMA, TPP, and autoTiering. We use additional enhancements to address the shortcomings of other mechanisms discussed in Section 2.5: (a) better estimation of access rate, and (b) a contextual (time-based) feature for page accesses with *burst duration*.

4.1 Page Table Poisoning

At an interval of one second, INDIGO periodically marks all pages of an active process to be inaccessible and records the *marking time* (M_i). If a page has already been marked in the current interval (but not accessed), it is not marked again in the next interval. When a process is accessing memory for the first time after the marking, a page fault (known as a *NUMA hinting fault* [1]) is generated that can be intercepted to record the corresponding *access time* (A_i). After each access, pages are unmarked to prevent additional faults in that interval. Using this method, one can generate a timeline of marking and access times for each page. We use that timeline to estimate the page access rate.

4.2 Point Access Rate Estimator

To estimate the page access rate, we use a combination of a pair of marking and access times. One can see intuitively that if a page is accessed immediately after it is marked (i.e., the difference between the access and marking times is small), its access rate will likely be high. Conversely, if a page is marked but not accessed for a long duration of time (i.e., the difference between the access and marking times is large), its access rate would be low.

$$F_i = 1/(A_i - M_i) \tag{2}$$

We use Equation (2) to estimate F_i, the access rate for the i-th marking and access time pair. Note that A_i will always be greater than M_i, as access occurs after marking.

4.3 Coalescing Access Measurements

Based on Insight 2 in Section 2.6, a single access rate measurement based on a moving average is insufficient to capture the temporal behavior of page access patterns. To capture the access patterns across time, we also group measurements with similar access frequencies together (i.e., a *burst*) to estimate the duration of access (i.e., the *burst duration*). We can use an online segmentation algorithm to perform this grouping. Algorithm 1 is one such segmentation algorithm that estimates the burst duration for page access; δ_1 denotes the closeness of two access rate measurements and δ_2 is simply the page marking interval. The values for δ_1 and δ_2 are selected with Bayesian optimization. For each page, we maintain an additional variable in the page struct to denote the current size of the cluster/burst. If the next access rate measurement is close in magnitude to the previous measurement and the measurement interval (i.e., the duration between two adjacent marking times) is small, then the next measurement is added to the current cluster. Otherwise, we create a new cluster and start collecting the next measurement. The size of each cluster effectively represents the burst duration based on the first and last measurements added in the cluster. Overall, the clustering output from the algorithm would be a tuple of the burst duration C and the estimated access rate F for every page across all measurements. The hyper-parameters in page telemetry such as page marking time and closeness thresholds are determined empirically.

As an alternative to page marking, PEBS counters can also be utilized to determine page access rates. However, INDIGO does not use PEBS because: (a) PEBS-based profiling leads to increased overheads for workloads with high memory bandwidth requirements [44], and (b) limited number of perf counters are supported in CPUs that may be required for other system monitoring tasks.

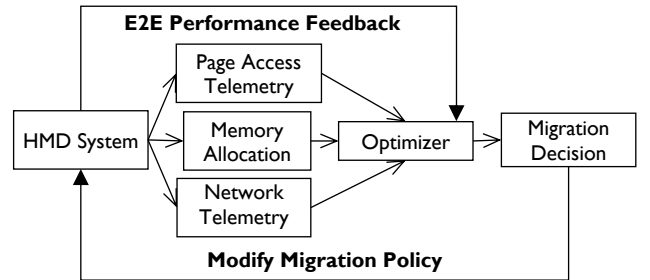## 5 Adaptive Page Migration

Figure 7: Feedback loop to solve the optimization problem for page migration.

In Insight 3 in Section 2.6, we showed that the optimal page migration policy is difficult to solve online due to the high computational complexity of O(pages^3). We present two alternative approaches that can be used, depending on training data availability: (a) a network adaptive migration policy (as described in Section 5.1), and (b) a contextual multi-armed bandit agent that learns optimal migration policies trained with application performance metrics via a feedback loop guided by network telemetry, page telemetry, and memory allocations (as described in Sections 5.2 and 5.3).

5.1 Network Adaptive Page Promotion

We present a network-adaptive page promotion policy that does not require training data, based on Equation (1). By rearranging terms in the equation, we determine that pages should be promoted if $(\text{access}_p - \text{access}_d) \times \text{Available Bandwidth} \times \Delta\text{Latency} > \text{threshold}$. The terms access_p and access_d (i.e., the access rates of the promoted and demoted pages) can be obtained at decision time using Page Telemetry, presented in Section 4.2. The terms Available Bandwidth and Δ Latency can also be obtained online using network hardware performance counters with low overhead.

While this policy is simple to implement within an existing framework such as TPP, a few limitations prevent it from achieving optimal performance. The policy does not take into account the dynamic nature of the application access patterns and adapt appropriately; the estimate of access_d is stale, as demotion is not on the critical path of promotion; and threshold selection is empirically guided (similar to other mechanisms, like Nimble and HeMem). To resolve these problems, we present the feedback loop approach with a contextual multi-armed bandit agent in Section 5.2.

5.2 Feedback Loop Formulation

Figure 7 shows a feedback loop that can be used to solve the decision problem involved in page migration. Again relying on Equation 1, we derive features necessary for input to the feedback loop. (a) *Page Access Telemetry:* access_p (the access rate of the promoted page) is estimated using page access telemetry. (b) *Memory Allocation:* access_d is dependent on the memory allocation. Larger values of memory allocation will lead to smaller access_d values and vice versa. (c) *Network Telemetry:* The network cost term ($\frac{\text{page size}}{\text{bandwidth} \times \Delta \text{ latency}}$) can be estimated from network telemetry (e.g., via network hardware performance counters). The migration decision computed by an optimizer simply decides whether the page should transfer from remote to local memory. Application performance is used as feedback to improve the optimizer over multiple training runs.

However, that feedback loop is still impractical to use, as the page migration decision is made on a *per-page* basis. The optimizer is therefore difficult to scale, as every page migration requires an inference on the optimizer. To resolve the problem, we changed the "Migration Decision" to a "Migration Threshold," which is common to all pages. Any page whose access rate is above the "Migration Threshold" is eligible for page promotion. The feedback loop is now scalable, as it is independent of the number of pages used by the application.

The choice of optimizer can be varied, and there are multiple possible options. We specifically chose contextual multi-armed bandits because (a) it requires less hyperparameter tuning than reinforcement learning algorithms, such as PPO [58] and DDPG [39]; and (b) the inference overhead is smaller than that of Bayesian optimization.

5.3 Feedback Loop Optimization

Solving the Feedback Loop. We model the solution to the feedback loop using a contextual multi-armed bandits (cMAB) agent. cMAB [42] are a class of online decision-making problems in which an agent must choose from among several actions or "arms" to optimize a *reward* given the current state of the environment defined as the *context*. Each episode is a single step corresponding to an

Hyperparameters	Value
num_hidden_layer	2
hidden_layer_size	64
learning_rate	0.0005
exploration_fraction	0.1
batch_size	32
MAX_TRAIN	100000

Table 2: cMAB agent training hyperparameters.

application run. We use Equation (1) to define the context, arms, and reward of the cMAB agent on a local node.

Context. The context to the cMAB agent is equivalent to the input to the feedback loop in Figure 7 that includes page telemetry, memory allocation, and network telemetry. By using migration thresholds (as discussed in Section 5.2), we shift page telemetry to the actions and not use it in the context. Therefore, our context consists of features relevant to memory allocation and network telemetry: (a) *Local Memory Allocation:* The total memory limit on the local node, which is limited by a user-defined `sysctl` variable. (b) *Local Node Memory Usage:* The maximum total memory used at the local node across the application run. (c) *Remote Memory Pool Usage:* The maximum total memory used at the remote node. (d) *Memory Network Traffic:* The total traffic sent on the memory network between the local node and remote memory pool.

Actions. The actions correspond to migration threshold selection that include: (a) *Burst Duration Threshold* and (b) *Access Rate Threshold*. Note that, an accessed page will be considered as a candidate for promotion only if the measured burst duration and access rate is above the corresponding threshold values.

Reward. We define the reward function for the cMAB agent using application performance as the primary optimization goal. Improving application performance is equivalent to maximizing the access locality benefits described in Equation (1). For batch jobs, we define application performance simply as the completion time. On the other hand, for request-serving applications that do not finish, we mock up their completion times using the inverse of requests served per second. The final reward is the negative of the application completion time, to penalize higher completion times.

cMAB Implementation and Training. We implemented the cMAB agent based on a Deep Q-learning Network (DQN) [63] algorithm with $\gamma = 0$ to mimic a contextual multi-armed bandits solver. The action space is discretized to use the thresholds as actions for the DQN solver. Table 2 shows the hyperparameters of the DQN agent used in our implementation and experiments with INDIGO. We selected the same hyperparameters as [46] for faster policy training convergence. We trained one cMAB agent with varying local and remote memory allocation configurations that ranged between 10% and 90% of the application's memory working set size. The application was first run multiple times with maximal local memory allocation, and the cMAB agent was trained for `MAX_TRAIN` episodes. Next, the local memory allocation was gradually decreased with a step size of 10% of the application working set size. Each allocation was allowed to be trained for `MAX_TRAIN` episodes during the training of the cMAB agent.

5.4 INDIGO Kernel Implementation

INDIGO promotes pages when NUMA hinting page faults arise (as described in Section 4). Upon receiving a page fault, we update the burst duration and get the access rate estimate using the clustering algorithm described in Section 4. The burst duration and access timestamps are recorded in the Linux kernel as an additional variable in `struct page` that uses 8 bytes. The overhead is negligible, as it corresponds to <0.1% increase in total memory usage. The burst duration and access rate are then compared against the threshold set by the cMAB agent as described in Section 5.2. In addition, we also implemented a `LOW_WATERMARK` variable that prevents further pages from being migrated if no space is available on the local node. The `LOW_WATERMARK` variable denotes a hard threshold to prevent the local node memory from exceeding its allocation. If a page satisfies the `LOW_WATERMARK` and meets the cMAB-set thresholds, an attempt is made to transfer the page from the remote node to the local node by reusing the existing `migrate_pages()` function provided by the Linux kernel tools for NUMA systems.

We reused the existing Linux disk paging implementation for page demotion using the `kswapd` daemon. Instead of swapping to the disk, pages are demoted to the remote node. We wake up the `kswapd` daemon if memory allocation on the local node exceeds a `HIGH_WATERMARK`. `HIGH_WATERMARK` is set approximately 10 MB higher than `LOW_WATERMARK` to allow for slack to enable page promotion if the local node's memory usage is reaching its allocation capacity.

6 Evaluation

Our evaluation aims to address the following aspects:

(a) Variance in performance gains from INDIGO across applications,
(b) INDIGO's performance in network contention scenarios,
(c) INDIGO's performance when pretrained on a similar application,
(d) INDIGO's training time and performance degradation during training,
(e) Breakdown of performance gains by each INDIGO component, and
(f) The overhead cost of running INDIGO along with the HMD system.

6.1 System Setup

We used ThymesisFlow [8], an open-source prototype for hardware memory disaggregation. ThymesisFlow implements a hardware-software co-designed memory disaggregation interconnect on top of the POWER9 architecture by directly interfacing the memory bus via the OpenCAPI cache-coherent interconnect protocol. Each node featured a dual-socket POWER9 CPU (32 physical cores and 128 parallel hardware threads) and 512 GB of RAM. Each node is equipped with an AlphaData 9V3 card that featured a Xilinx Ultrascale FPGA that implemented the OpenCAPI stack and the ThymesisFlow logic. Our prototype implements the HMD system described in Section 2.1, in which the memory network has a shared 100 Gb/s link to the remote memory pool with ~900ns access latency [52]. Network contention is observed on this shared link.

Application	Memory Usage	Description
Redis [16]	16 GB	In-memory data structure store, run with memtier benchmark with 10,000 requests/s
Graph500 [49]	48 GB	Multiple iterations of Breadth First Search (BFS) and Single Source Shortest Path (SSSP), configuration uses problem size = 25, edge factor = 16
voltDB [60]	12 GB	In-memory and fully ACID-compliant RDBMS with a share-nothing architecture, run with TPC-C benchmark
HashJoin [69]	36 GB	Database join algorithm that uses hashcodes in equijoins
ML Inference [64]	100 GB	Transformer-based model inference
AMG [61]	40 GB	Parallel algebraic multigrid solver that represents common operations in HPC systems

Table 3: Applications and their configurations used in evaluating INDIGO.

6.2 Experiment and Workload Setup

To evaluate INDIGO, we selected the memory-intensive applications shown in Table 3. These applications are representative of common workloads in cloud datacenters or HPC systems.

Experiment Methodology. We ran experiments with five page migration mechanisms: INDIGO, INDIGO Network Adaptive Policy (which does not require training), MEMTIS, TPP, and Nimble. The applications were run with different local-to-remote memory allocation ratios varying between 10–90%. In addition to these page migration mechanisms, we also ran the applications in the "None" configuration which denotes only remote memory was used. Contention on the memory disaggregation fabric was modeled by executing the STREAM benchmark concurrently with applications. INDIGO was trained on each application separately, using the contextual multi-armed bandit agent described in Section 5.2. To train each application, we decremented the local memory allocation by 10% after every `MAX_TRAIN` episodes. The agent learned and adjusted its actions (the cutoff for burst duration and access rate) for each new allocation. To avoid unnecessary application execution during training, we also cached the context-to-reward evaluation. Specifically, we cache the mapping from the context, access rate, and burst duration cutoffs to application runtime after application execution. When the same context and action set was encountered again, we could simply retrieve it from the saved mapping instead of rerunning the application. Without the caching mechanism, the cMAB training algorithm would run each application for `MAX_TRAIN` episodes. However, the introduction of caching reduced the number of application runs to ~1000, resulting in a ~100× reduction in training time.

6.3 Evaluation Results

Application Performance. We find that the performance benefit from page migration depends greatly on the application characteristics and the local-to-remote memory allocation ratio. Figures 8 and 9 show the variation in applications' runtime degradation when they were running with INDIGO, Nimble, TPP, MEMTIS, and page migration completely disabled. We define runtime degradation as
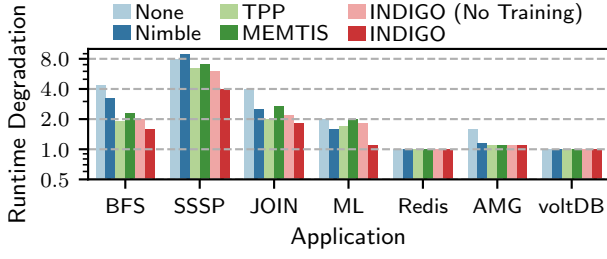
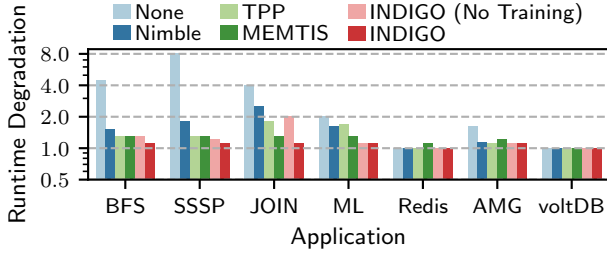**Figure 8: Per-application runtime degradation across page migration algorithms for 10% local memory allocation.**

Figure 9: Per-application runtime degradation across page migration algorithms for 80% local memory allocation.
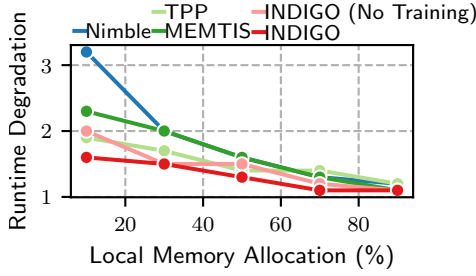

Figure 10: BFS runtime degradation with increasing local memory allocations for different page migration mechanisms.

the ratio between the observed runtime and the runtime with 100% local memory allocation (i.e., the lowest possible runtime for the application). Each application was run in two configurations: (a) with low local memory (10% of the total application working set size), and (b) with high local memory (80% of the total size).

In addition, our evaluation confirms several hypotheses about the relationship among page migration performance, memory allocations, and application characteristics.

• Overall, we find that INDIGO offers a runtime improvement of up to 50–70% compared to other page migration mechanisms. Runtime with INDIGO (No Training) is comparable to other mechanisms without network contention, but offers up to 30–50% improvement under network contention (as we describe later in the evaluation).

• Higher local memory allocations result in less runtime degradation. For example, when comparing Figure 8 and Figure 9 across applications, we find that increasing the local memory allocation reduces runtime degradation. The lower runtime can be explained

by the smaller average memory access latency as additional pages are placed in local memory.

• Applications with high computational overhead, such as Redis[4] and voltDB, do not experience significant performance degradation from the use of remote memory. In consequence, none of the page migration mechanisms are able to provide a significant benefit, as these applications are mostly computing on CPU rather than accessing memory.

• While page migration with INDIGO can reduce runtime degradation significantly, it may not be sufficient to close the gap between the baseline (full local memory allocation) and partial local memory allocation. This is especially true for applications with more randomized access patterns, such as graph traversal (BFS and SSSP). However, given the constraint of limited local memory, INDIGO still outperforms other mechanisms.

• We find that for numerous applications and local memory allocation combinations, application performance can reach nearly the baseline. Such improvement is observed when the application performs significant computation on the migrated pages (for e.g. layer-by-layer inference in case of ML), hence appropriate page migration is sufficient to completely hide remote access latency.

• Figure 10 shows that INDIGO's advantage over other page migration mechanisms, such as Nimble, TPP, and MEMTIS, is more pronounced for smaller local memory allocations. This observation can be explained by the left side of Equation (1). As the local memory allocation decreases, the access rate of the demoted page ($access_d$) increases, and the corresponding optimality threshold of $access_p$ also increases. Therefore, fewer pages benefit from promotion, and INDIGO's selectivity in page promotion becomes critical. Specifically, we find that INDIGO promotes between 20–40% less pages compared to other page migration mechanisms.
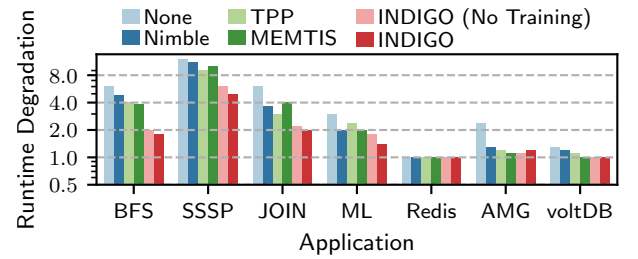

Figure 11: Per-application runtime degradation during network contention for 10% local memory allocation.

Impact of Network Contention. To understand the impact of network contention on the migration mechanisms, we used the STREAM benchmark (that performs serial memory copy operations via the COPY kernel) to produce memory traffic. STREAM combined with other applications generates network contention by utilizing 50% of the available link bandwidth. Such network contention corresponds to bandwidth contention observed in commercial HMD systems [5]. Figure 11 shows the impact on runtime degradation for each application and page migration mechanism

---

[4]Redis communicates via TCP sockets which becomes the dominant cost, and remote memory access latency is effectively hidden.
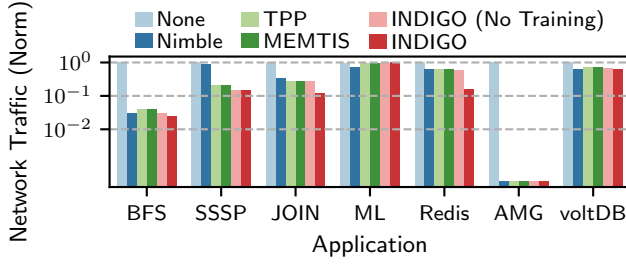
Figure 12: Per-application network traffic across page migration algorithms for 10% local memory allocation.

under such network contention. We found that both versions of INDIGO performed better than other mechanisms. In particular, we observed that the performance of INDIGO (No Training) was significantly better than that of TPP, Nimble, and MEMTIS. For instance, the performance of the BFS application INDIGO (No Training) was comparable to that of TPP without network contention but provided a 50% performance improvement under network contention. The improved performance of both versions of INDIGO can be explained by the right side of Equation (1). INDIGO considers the network cost term ($\frac{\text{page size}}{\text{bandwidth} \times \Delta \text{ latency}}$) and lowers the promotion rate by setting a higher cutoff for the access_p estimate, thereby satisfying the optimal migration condition. For example, when running BFS, INDIGO transfers 12000 pages/sec under contention, compared to TPP that transfers 22000 pages/sec. Promotion rates for other mechanisms are shown in Figure 4.

Network Traffic Reduction. We found that page migration using INDIGO results in less total network traffic across applications than other page migration mechanisms. Figure 12 shows the normalized network traffic (cache miss and page migration traffic) for different applications and page migration algorithms with local memory allocation equal to 10% of the total application working set size. Traffic is normalized with respect to the full remote memory allocation that consumes the most network bandwidth. Total network traffic using INDIGO is up to 4000× lower (in case of AMG) and 2.3× lower (in case of JOIN) than can be achieved using only remote memory and other page migration algorithms. Again, the reason is the cumulative effect of traffic from page promotion and cache miss. Although the page promotion traffic is higher, the total number of cache misses decreases as fewer pages are placed in local memory. Hence, the overall network traffic (a combination of migration and access traffic) is lower with INDIGO. In addition, we found that the applications with the highest total network traffic may not have the worst runtime degradation, as the memory access cost can be masked by high computational costs (for example, due to expensive matrix multiplications in ML inference workloads or high network processing costs in the case of Redis).

Ablation Study of INDIGO Components. We performed an ablation study to understand the contribution of each INDIGO component on end-to-end performance improvement. To do so, we modified INDIGO by disabling each component one at a time and comparing the drop in performance relative to the original mechanism. The four components we identified for the ablation study were *burst duration*, which accounts for dynamic application access
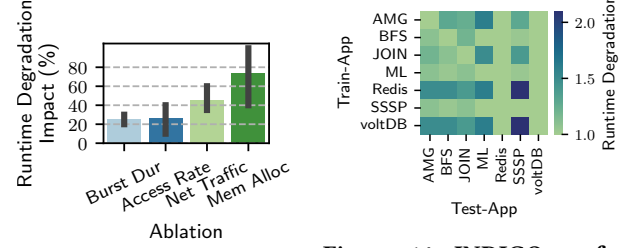

Figure 13: Ablation study of INDIGO components.

Figure 14: INDIGO performance transfer across applications without retraining.

Applications	INDIGO	Nimble	TPP	MEMTIS
SSSP	**6.2×**	9×	7.3×	7×
JOIN	**2.5×**	3.4×	3.2×	2.8×
voltDB	1×	1×	1×	1×

Table 4: Application runtime degradation across page migration algorithms in a multi-tenant setting.

patterns and decision context (ΔT); *access rate*, which accounts for the access rate estimate (access_p and access_d); *network traffic*, which accounts for the network cost term ($\frac{\text{page size}}{\text{bandwidth} \times \Delta \text{ latency}}$); and *memory allocation*, which, along with the access rate of the promoted page (access_p), can improve the estimate of the access rate of the demoted page (access_d). Figure 13 shows the reduction in gains due to the use of INDIGO, when each component is disabled. We found that all INDIGO components contribute significantly to overall performance gains, with memory allocation being the most significant.

Multi-tenant Performance. INDIGO can be extended to work in the multi-tenant setting where multiple applications on the local node are simultaneously migrating pages from the remote memory pool. There are two aspects to the INDIGO's extension in the multi-tenant setting: (a) the underlying network cost model from Equation (1), and (b) the Linux kernel implementation. The cost model continues to hold for each application's page migration performance with modifications in the bandwidth and Δlatency terms to account for contention between applications. Therefore, we can use the heuristic policy and trained cMAB model with per-application memory traffic measurements. The Linux kernel also needs to be modified to support per-application memory allocation, memory usage, and thresholds for access rate and burst duration. We support these per-application metrics by creating an additional task-metric mapping using the task struct within the kernel space.

To evaluate INDIGO in the multi-tenant setting, we choose the following three applications: (a) SSSP, (b) JOIN, and (c) voltDB. We choose SSSP as it is the most performance sensitive to page migration. voltDB and JOIN were chosen as they were the only other applications that prevent CPU core oversubscription. We avoid CPU core overscubscription, as in the event of oversubscription, page migration would not be the major cause of performance degradation. All applications were run with 10% local memory allocation. Table 4 shows the runtime degradation across page migration algorithms compared to an isolated full local memory allocation case. Overall, we find that the performance benefits of INDIGO continue to hold.

cMAB Transfer across Applications. We found that INDIGO performance transferred without retraining across similar applications. Figure 14 shows the runtime degradation when INDIGO was trained on the *Train-App* and tested against the *Test-App*. We found that for similar applications, such as BFS and SSSP, the performance of a trained cMAB agent transferred with less than 5% degradation. On the other hand, applications that have low variability in performance (such as voltDB and Redis) are bad candidates for training, as they do not allow any useful learning. In addition, we note that use of a pre-trained application model limited the maximum amount of degradation to only twice that of the baseline degradation. Thus, a pre-trained model can be used to kickstart the re-training and avoid high initial runtime degradation.

Runtime Overhead. INDIGO introduces low runtime and memory overhead compared to complete disabling of page migration. Runtime overhead is primarily introduced from the extra computational cost associated with page marking and page faults and in the worst-case is limited to 10% of the application runtime in our experiments. The runtime overhead is not specific to INDIGO, as other page migration mechanisms, such as TPP and Nimble, use identical page poisoning systems. Memory overhead is introduced by modifying the page struct data structure to record the burst duration. The memory overhead corresponds to an increase of 8 bytes for every page in the operating system (i.e., 8 bytes for 64 KB ~0.01%).

7 Discussion

Difference between CXL and OpenCAPI. INDIGO uses ThymesisFlow [8], which instantiates an HMD system using the OpenCAPI coherent interconnect protocol. However, we expect that most future research activity targeting system composability or component disaggregation will focus on CXL because the OpenCAPI consortium has donated all of its IPs to the CXL Consortium [4]. OpenCAPI [7] and CXL [3] have some fundamental physical differences but share most of the concepts related to memory access and operating system support. Both OpenCAPI and CXL enable manipulation of memory external to a machine by means of a dedicated CPU-less NUMA node, alongside local NUMA nodes hosting one or more processors and their local memory. Also, in both cases, remote memory is accessed at the cache line granularity. Therefore, INDIGO and any page migration mechanism that moves pages across NUMA nodes are applicable to both technologies.

INDIGO Scalability to Multiple Remote Memory Pools. INDIGO considers all remote memory as a single NUMA node and relies on the hardware implementation to implement multiple remote memory pools. In addition, the degradation in access latency when moving from local to remote memory is 10−100× [25, 50, 52], which accounts for a much larger slowdown than that caused by variation within remote memory pools. Hence, we expect that the performance of INDIGO will not be significantly affected by the choice of remote memory pool that will be the candidate for page promotion and demotion.

8 Related Work

Page Migration. Prior work has explored hardware-assisted [37, 45, 47, 54] and application-guided [18, 23, 67] page migration for

tiered memory systems, which may not be practical as they require specialized hardware support or application redesign from the ground up. A large body of work exists in page migration for disaggregated and tiered memory that is independent of the application and hardware support [10, 17, 22, 28, 29, 41, 43, 44, 48, 56, 68]. We discuss the limitations of these approaches in Section 2.5.

Memory Disaggregation. Memory disaggregation exposes capacity available in remote hosts as a pool of memory shared among many machines. OS-based [12, 13, 26, 59] or runtime-based memory disaggregation [11, 57, 66] is designed for RDMA over InfiniBand or Ethernet networks. However, the access latency is an order of magnitude higher than for hardware-based approaches. Recently, with the introduction of interconnect protocols, there has been a rise in HMD systems that directly connect the processor and remote memory via the cache-line such as Pond [35], DirectCXL [25], and ThymesisFlow [52]. INDIGO can be used in conjunction with such HMD systems to mitigate the impact of remote memory access latency.

9 Conclusion

We present INDIGO, a network-aware page migration framework for HMD systems. At the core of INDIGO is an adaptive page promoter that uses novel page telemetry metrics, such as access rate and burst duration, to select pages dynamically for promotion from remote to local memory via a contextual multi-armed bandit agent. Evaluation of INDIGO on a real HMD system with common cloud and HPC workloads shows up to 50−70% improvement in application performance. INDIGO and further research on page migration is crucial for the reduction of the TCO of infrastructure, by enabling data center providers to deploy cost-effective and high-capacity shared memory pools.

References

[1] [n. d.]. autoNUMA for tiered memory. https://lwn.net/Articles/835402/. Accessed: Feb 2023.

[2] [n. d.]. Cache Coherent Interconnect for Accelerators. https://www.ccixconsortium.com/. Accessed: 2021-07-24.

[3] [n. d.]. Compute Express Link. https://www.computeexpresslink.org/. Accessed: 2021-07-24.

[4] [n. d.]. CXL Consortium and OpenCAPI Consortium Sign Letter of Intent to Transfer OpenCAPI Specifications to CXL . https://www.computeexpresslink.org/_files/ugd/0c1418_d3474155dc6e4929aa2a5658a894d1a6.pdf. Accessed: Apr 2023.

[5] [n. d.]. H3 Platform. https://www.h3platform.com/. Accessed: July 2024.

[6] [n. d.]. Linux kernel - Page migration. https://www.kernel.org/doc/html/v5.0/vm/page_migration.html. Accessed: Feb 2023.

[7] [n. d.]. OpenCAPI. https://opencapi.org/. Accessed: 2021-07-24.

[8] [n. d.]. ThymesisFlow Home Page. https://github.com/OpenCAPI/ThymesisFlow/. Accessed: 2022-02-08.

[9] [n. d.]. xConn CXL 2.0 Switch. https://www.prnewswire.com/news-releases/xconn-technologies-releases-early-production-silicon-samples-of-apollo-cxl-2-0-switch-at-memcon-2024--302100130.html. Accessed: July 2024.

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

[11] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. 2018. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 775–787.

[12] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 843–857.

[13] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker.

2020. Can far memory improve job throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[14] Lars Bergstrom. 2011. Measuring NUMA effects with the STREAM benchmark. *arXiv preprint arXiv:1103.3225* (2011).

[15] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 79–92.

[16] Josiah Carlson. 2013. *Redis in action*. Simon and Schuster.

[17] Thaleia Dimitra Doudali, Daniel Zahka, and Ada Gavrilovska. 2021. Cori: Dancing to the right beat of periodic data movements over hybrid memory systems. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 350–359.

[18] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[19] Michael J Feeley, William E Morgan, EP Pighin, Anna R Karlin, Henry M Levy, and Chandramohan A Thekkath. 1995. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM symposium on Operating Systems Principles*. 201–212.

[20] Hormozd Gahvari, Allison H Baker, Martin Schulz, Ulrike Meier Yang, Kirk E Jordan, and William Gropp. 2011. Modeling the performance of an algebraic multigrid cycle on HPC platforms. In *Proceedings of the international conference on Supercomputing*. 172–181.

[21] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation.. In *OSDI*, Vol. 16. 249–264.

[22] Christina Giannoula, Kailong Huang, Jonathan Tang, Nectarios Koziris, Georgios Goumas, Zeshan Chishti, and Nandita Vijaykumar. 2023. DaeMon: Architectural Support for Efficient Data Movement in Fully Disaggregated Systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7, 1 (2023), 1–36.

[23] Michael Giardino, Kshitij Doshi, and Bonnie Ferri. 2016. Soft2LM: Application guided heterogeneous memory management. In *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 1–10.

[24] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory Pooling with CXL. *IEEE Micro* (2023).

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

[26] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with InfiniSwap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[27] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. Heteroos: OS design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 521–534.

[28] Hyunik Kim, Changyeon Jo, Jörn Altmann, and Bernhard Egger. 2021. RapidSwap: a Hierarchical Far Memory. In *Economics of Grids, Clouds, Systems, and Services: 18th International Conference, GECON 2021, Virtual Event, September 21–23, 2021, Proceedings 18*. Springer, 143–151.

[29] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems.. In *USENIX Annual Technical Conference*. 715–728.

[30] Samir Koussih, Anurag Acharya, and Sanjeev Setia. 1999. Dodo: A user-level system for exploiting idle memory in workstation clusters. In *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No. 99TH8469)*. IEEE, 301–308.

[31] H. W. Kuhn. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly* 2, 1-2 (1955), 83–97. https://doi.org/10.1002/nav.3800020109 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nav.3800020109

[32] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. 2016. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 705–721.

[33] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Memtis: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[34] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G Shin. 2022. Hydra: Resilient and highly available remote memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 181–198.

[35] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 574–587.

[36] Haifeng Li, Ke Liu, Ting Liang, Zuojun Li, Tianyue Lu, Hui Yuan, Yinben Xia, Yungang Bao, Mingyu Chen, and Yizhou Shan. 2023. HoPP: Hardware-Software Co-Designed Page Prefetching for Disaggregated Memory. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1168–1181.

[37] Haifeng Li, Ke Liu, Ting Liang, Zuojun Li, Tianyue Lu, Hui Yuan, Yinben Xia, Yungang Bao, Mingyu Chen, and Yizhou Shan. 2023. HoPP: Hardware-Software Co-Designed Page Prefetching for Disaggregated Memory. *arXiv preprint* (2023).

[38] Shuang Liang, Ranjit Noronha, and Dhabaleswar K Panda. 2005. Swapping to remote memory over infiniband: An approach using a high performance network block device. In *2005 IEEE International Conference on Cluster Computing*. IEEE, 1–10.

[39] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning. In *Proceedings of the 4th International Conference on Learning Representations (ICLR 2016)* (San Juan, Puerto Rico), Yoshua Bengio and Yann LeCun (Eds.). https://arxiv.org/abs/1509.02971.

[40] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH computer architecture news* 37, 3 (2009), 267–278.

[41] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. 2019. Hierarchical hybrid memory management in OS for tiered memory systems. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2223–2236.

[42] Tyler Lu, Dávid Pál, and Martin Pál. 2010. Contextual multi-armed bandits. In *Proceedings of the Thirteenth international conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, 485–492.

[43] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. 2022. MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems.. In *HPCA*. 925–937.

[44] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 742–755. https://doi.org/10.1145/3582016.3582063

[45] Mitesh R Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H Loh. 2015. 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)*. IEEE, 126–136.

[46] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.

[47] Jeffrey C Mogul, Eduardo Argollo, Mehul A Shah, and Paolo Faraboschi. 2009. Operating System Support for NVM+ DRAM Hybrid Main Memory.. In *HotOS*, Vol. 9. 4–14.

[48] Yaebin Moon, Wanju Doh, Kwanhee Kyung, Eojin Lee, and Jung Ho Ahn. 2023. ADT: Aggressive Demotion and Promotion for Tiered Memory. *IEEE Computer Architecture Letters* (2023).

[49] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[50] Archit Patke, Haoran Qiu, Saurabh Jha, Srikumar Venugopal, Michele Gazzetti, Christian Pinto, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2022. Evaluating Hardware Memory Disaggregation under Delay and Contention. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1221–1227.

[51] Vinicius Petrucci, Eishan Mirakhur, Rita Gupta, Mahesh Wagh, Nikesh Agarwal, Su Wei Lim, and Vishal Tanna. 2024. CXL Memory Expansion: A Closer Look on Actual Platform. (2024). https://www.micron.com/content/dam/micron/global/public/products/white-paper/cxl-memory-expansion-a-close-look-on-actual-platform.pdf.

[52] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H Peter Hofstee. 2020. ThymesisFlow: a software-defined, HW/SW co-designed interconnect stack for rack-scale memory disaggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 868–880.

[53] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. 2023. Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 181–198.

[54] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*. 85–95.

[55] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 392–407.

[56] Jie Ren, Dong Xu, Ivy Peng, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2023. HM-Keeper: Scalable Page Management for Multi-Tiered Large Memory Systems. *arXiv preprint arXiv:2302.09468* (2023).

[57] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*. 315–332.

[58] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017), 12 pages. http://arxiv.org/abs/1707.06347

[59] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*. 69–87.

[60] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.

[61] Klaus Stüben. 1983. Algebraic multigrid (AMG): experiences and comparisons. *Applied mathematics and computation* 13, 3-4 (1983), 419–451.

[62] Dimitris Syrivelis, Andrea Reale, Kostas Katrinis, Ilias Syrigos, Maciej Bielski, Dimitris Theodoropoulos, Dionisios N Pnevmatikatos, and Georgios Zervas. 2017. A software-defined architecture and prototype for disaggregated memory rack scale systems. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, 300–307.

[63] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double Q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 30.

[64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[65] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating system support for improving data locality on CC-NUMA compute servers. *ACM SIGOPS Operating Systems Review* 30, 5 (1996), 279–289.

[66] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A Memory-Disaggregated Managed Runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*. 261–280.

[67] Wei Wei, Dejun Jiang, Sally A McKee, Jin Xiong, and Mingyu Chen. 2015. Exploiting program semantics to place data in hybrid memory. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 163–173.

[68] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 331–345.

[69] Hansjörg Zeller and Jim Gray. 1990. An Adaptive Hash Join Algorithm for Multiuser Environments.. In *VLDB*, Vol. 90. 186–197.

[70] Yang Zhou, Hassan MG Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E Culler, Henry M Levy, et al. 2022. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*. 55–71.