THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Reconsidering OS Memory Optimizations in the Presence of Disaggregated Memory

Citation for published version:
Bergman, S, Faldu, P, Grot, B, Vilanova, L & Silberstein, M 2022, Reconsidering OS Memory Optimizations in the Presence of Disaggregated Memory. in M Lippautz & D Chisnall (eds), *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management.* ACM, pp. 1-14, 2022 ACM SIGPLAN International Symposium on Memory Management, San Diego, California, United States, 14/06/22. https://doi.org/10.1145/3520263.3534650

Digital Object Identifier (DOI):
10.1145/3520263.3534650

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 22. Jul. 2022

Reconsidering OS Memory Optimizations in the Presence of Disaggregated Memory

Shai Bergman[1] Priyank Faldu[2*] Boris Grot[3] Lluís Vilanova[4] Mark Silberstein[1]

[1]Technion - Israel Institute of Technology [2]ARM [3]University of Edinburgh [4]Imperial College London

Abstract

Tiered memory systems introduce an additional memory level with higher-than-local-DRAM access latency and require sophisticated memory management mechanisms to achieve cost-efficiency and high performance. Recent works focus on byte-addressable tiered memory architectures which offer better performance than pure swap-based systems. We observe that adding disaggregation to a byte-addressable tiered memory architecture requires important design changes that deviate from the common techniques that target lower-latency non-volatile memory systems. Our comprehensive analysis of real workloads shows that the high access latency to disaggregated memory undermines the utility of well-established memory management optimizations. Based on these insights, we develop HotBox – a disaggregated memory management subsystem for Linux that strives to maximize the local memory hit rate with low memory management overhead. HotBox introduces only minor changes to the Linux kernel while outperforming state-of-the-art systems on memory-intensive benchmarks by up to 2.25×.

CCS Concepts: • **Software and its engineering** → **Memory management**; • **Hardware** → **Emerging architectures**.

Keywords: Disaggregated Memory, Operating Systems

ACM Reference Format:

Shai Bergman, Priyank Faldu, Boris Grot, Lluís Vilanova, and Mark Silberstein. 2022. Reconsidering OS Memory Optimizations in the Presence of Disaggregated Memory. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management (ISMM '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3520263.3534650

*Work done during the author's PhD studies at the University of Edinburgh.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISMM '22, June 14, 2022, San Diego, CA, USA

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9267-9/22/06…$15.00
https://doi.org/10.1145/3520263.3534650

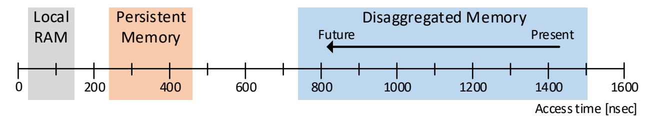


Figure 1. Latency ranges of tiered memory technologies.

1 Introduction

Tiered, or heterogeneous, memory architectures have emerged as a promising approach for delivering higher memory capacity at a lower cost. Such architectures extend the memory hierarchy with an additional memory *tier* that has a lower per-byte cost but also a higher access latency.

Two primary technologies are used in tiered memory systems, shown in Figure 1, which can be categorized according to their access latency. The first, local byte-addressable non-volatile memory (NVM), such as Intel Optane DC DIMMs, is a new type of memory hardware that is 2×– 4× slower than DRAM [27, 49], but cheaper by about the same factor.

The second, remote *disaggregated* memory, is a system architecture that delivers high memory capacity at a rack- or cluster-level over the network. Figure 2 shows the canonical disaggregated memory architecture where applications run on compute blades that host a small amount of *local* main memory, while the bulk of their datasets reside in a high-capacity *remote memory* blade accessed through a low-latency interconnect [33]. Disaggregation brings cost savings by reducing memory fragmentation across datacenter nodes, which share a large memory pool.

There are two mechanisms to transparently access disaggregated memory: using it as a swap device with accesses at page granularity, or via CPU load/store accesses at cache-line granularity. A hybrid approach combines cache-line accesses with *page migration* between remote and local memory, using the latter as a cache. Swapping can be employed in commodity systems today and has been thoroughly studied in prior work [10, 22, 24, 42]. However, a hybrid approach has gained increased interest [7, 15, 28, 48] with the expected emergence of lower latency networks offering end-to-end latencies as low as 750 nsec for optimistic mid-term estimates [2, 8, 22, 43] (accounting for network delays in the cache-coherent fabric, e.g., CXL [3] and Gen-Z [2], and local access in memory blades), to 1,500 nsec with today's interconnects [4, 13].

In this paper, we seek to optimize the performance of disaggregated memory systems with a hybrid access mechanism.

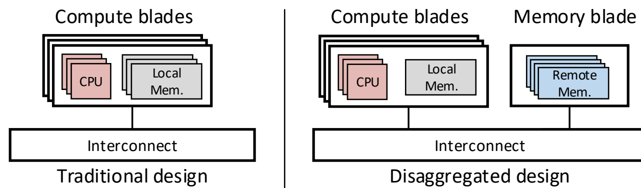


Figure 2. Traditional vs disaggregated memory architectures.

The key question shared by all tiered memory systems is how to reduce memory costs by shifting memory usage into the slower and cheaper tier with minimal impact to the application performance. Most existing mechanisms for managing byte-addressable tiered memory target the lower-latency end of the spectrum of 200–600 nsec [7, 20, 26, 28, 48]. In this work, we ask *whether the performance optimizations suggested in prior work are also effective for byte-addressable disaggregated memory management*, under their respective latency spectrum of 750–1500 nsec. Our analysis (§ 3) shows that this is not the case. Our conclusions are as follows:

1. A swap-only approach under higher memory latencies is inefficient. Several prior works [10, 22, 24, 42] regard the disaggregated memory blade as a swap device, whereby any access to data residing in the blade results in the page being copied to local DRAM. This approach leads to large-granularity accesses resulting in thrashing of local memory and poor performance on workloads with poor access locality. Our analysis shows that, for disaggregated memory latencies, such a swap-only approach can be inferior to a *hybrid* memory access mechanism that combines both cache-granular and page-granular accesses, even when using swap with an optimal offline page replacement policy.

2. Huge pages are detrimental. While prior approaches [7, 48] have advocated for using huge pages in tiered memory systems to reduce TLB pressure and shorten page walk latencies, we find that in a higher-latency regime, huge pages often have a *detrimental effect* on system performance. Our analysis shows the negative impact of a huge-page-induced phenomenon we call *hotness fragmentation*. Hotness fragmentation of a page implies that its constituting *base pages* are accessed at a different frequency (have different hotness)[1]. When migrated into local memory, hotness-fragmented huge pages decrease the effective local memory capacity and, as a result, increase the incidence of slow accesses to disaggregated memory.

3. Batch-migration of pages is inefficient. Prior studies [48] argued for migrating batches of pages to amortize system overheads. However, we find that batching decreases both the accuracy and timeliness of migration decisions, while at the same time does not seem to reduce the migration costs.

[1]Unlike the known problem of memory bloat, which stems from the internal fragmentation of huge pages due to non-contiguous allocations, hotness fragmentation is caused by mixed spatial access locality within a huge page.

To summarize, swapping, huge pages, and batching are well-established solutions to improve performance in tiered memory systems, but do so at the expense of reducing local memory hit rate with their *coarse-granular migration policy and mechanism*. This is a very profitable trade-off in existing tiered memory systems at the extremes of the latency differences between the fast and slow memories (e.g., local DRAM and NVM in the left half of Figure 1), but we demonstrate that this trade-off requires a different system design point for disaggregated memory because of its particular access latency regime.

Based on these insights, we build **HotBox**, a novel memory management subsystem for the Linux kernel targeting byte-addressable disaggregated memory that *maximizes local memory hit rate through the use of finer-grain management mechanisms*. While maximizing hit rate is a common goal of all caching architectures, the primary challenge we cope with in disaggregated memory systems are their non-trivial migration and access monitoring overheads.

To this end, HotBox makes the following design choices: (1) use a hybrid access mechanism, (2) eliminate the negative effects of hotness fragmentation by using only base pages in local and disaggregated memory, and (3) do not use batching and instead migrate pages on demand, one page at a time.

HotBox takes several steps to reduce the costs of disaggregated memory management. It reduces the overheads of estimating local memory page hotness by using a *dynamic page access sampling* mechanism whose frequency increases with local memory pressure. This allows HotBox to accurately discriminate between pages of high hotness, leading to better *eviction* decisions for the local memory, exactly when it is particularly important for caching performance [40], yet keeps the sampling overheads low in the common case. To reduce sampling overheads for remote memory, HotBox takes advantage of the hierarchical translation structures, dismissing large memory regions where not a single page has been accessed (i.e., a sub-tree of the page table). Finally, HotBox monitors the *utility of migrations*, pausing migrations when deemed unnecessary (i.e., to prevent local memory thrashing).

We implement HotBox in Linux and evaluate it using Intel's Persistent Hybrid Memory Emulation Platform (PMEP) [19]. We compare HotBox with state-of-the-art mechanisms, including swapping-based systems such as InfiniSwap [10, 22, 24], as well as the recent Nimble [48] system for tiered memory, which relies on huge pages and batch migration to amortize page management overheads. We run memory-intensive applications including database (`VoltDB`), key-value store (`memcached`), and graph analytics (`Ligra` and `Graph500`). HotBox outperforms all state-of-the-art approaches, corroborating the conclusions of our earlier analysis. Compared to swapping-based systems and Nimble, HotBox achieves speedups of up to 4.5× and 2.25× respectively.

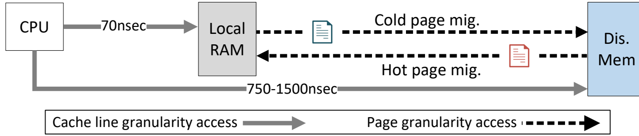


Figure 3. Disaggregated memory system model. CPU can perform direct cache line access to disaggregated memory. Hot and cold pages are migrated between the memory tiers.

2 Background

Disaggregated memory systems expose an additional memory tier into the traditional system architecture. The memory hierarchy of such a system consists of a fast but limited-capacity local DRAM, and a slower disaggregated memory blade. Figure 3 illustrates a hybrid disaggregated memory system in action. To attain maximum performance, frequently accessed (hot) pages are placed in the local DRAM, while less frequently accessed (cold) pages are placed in the disaggregated memory blade and accessed at cache line granularity over an interconnect fabric [2, 3, 16]. A page's hotness can change over time, and thus it is migrated between the tiers at runtime.

Recent works on tiered memory systems [7, 26, 28, 48] combine direct cache-line accesses to tiered memory and page migrations to optimize performance – that is, a hybrid system. They rely on mechanisms that sample page access frequency to determine hot and cold pages and migrate them accordingly. Cold pages are directly accessed from the tiered memory without migrating them first into local DRAM. These works, however, target low-latency tiered memory systems and are ill-suited for disaggregated memory systems as we show next.

3 Analysis of existing approaches

We identify three aspects of tiered memory systems that are essential for performance: (1) the granularity of memory accesses to remote memory (i.e., swapping vs cache line accesses vs a hybrid), (2) the granularity of memory management (i.e., page size), and (3) the granularity at which pages are selected and migrated between local and remote memory. In this section, we revisit the conclusions presented in the recent literature concerning these three issues and show that the reported findings are not applicable to the higher latency of disaggregated memory systems.

Methodology. Unless stated otherwise, we use several memory-intensive workloads, each of which has a working set size of about 10 GB, and analyze them on an evaluation platform with configurable latency for the disaggregated memory tier (see § 6 for details).

For brevity, we use the terms "local memory" and "remote memory" to refer to the local DRAM and disaggregated memories shown in Figure 3, respectively. In a tiered memory system without disaggregation, we use the term "remote memory" to refer to the slower memory tier (e.g., NVM).

LLC Cache	6 MB, 8-way, 64 B cache block, latency: 30 cycles
Local Mem.	1 GB, 4 KB page size, latency: 210 cycles
Remote Mem.	Unlimited size, 4 KB page size

Table 1. Disaggregated memory simulator parameters.

3.1 Granularity of memory accesses: cache line vs. swap

Prior works on transparent support for tiered memory architectures define the system as following one of three models:

Swap: Remote memory is used as a swap device [9, 10, 22, 24, 42].

Cache-line access: Data in remote memory is accessed directly using regular cache-line accesses [37].

Hybrid: A combination of the prior two models, as shown in Figure 3, that allows both direct access to remote memory at cache line granularity and page migration to local memory for performance optimizations [7, 15, 20, 28, 48].

Even though these approaches have been investigated independently in prior works, it remains unclear whether one model is superior to the others under disaggregated memory latencies. In this subsection, we confirm that the hybrid model is indeed essential for achieving high performance under these latencies: neither the swap nor cache-line access model in isolation is superior to the other one across all workloads.

We first consider the swap and cache line access models and show two applications where each model results in strictly poorer performance than the other. To do so, we compare regular cache line accesses against the *optimal swap model* using Belady's offline page replacement algorithm [12].

Analysis. We compare the models' average memory access time (AMAT) using a simulator. The simulator processes an application trace and computes AMAT for a specific cache size, for local memory and remote memory configurations.

We collect traces, using PIN [34], of two graph processing applications: `BFS` (large working set, mostly random accesses) and `Page Rank` (PR) (small working set) implemented in `Ligra` [44]. We choose the simulation parameters (Table 1) to match the scaled-down characteristics of a real system that we use for evaluation.

Figure 4 shows the AMAT slowdown of the two models over a configuration with only local memory (our ideal target), using various remote memory latencies (a lower slowdown implies better performance); note that we simulated both the optimal offline swap policy (based on Belady) as well as an online swap policy (similar to the one in Linux). The key result is that none of the access models outperforms the

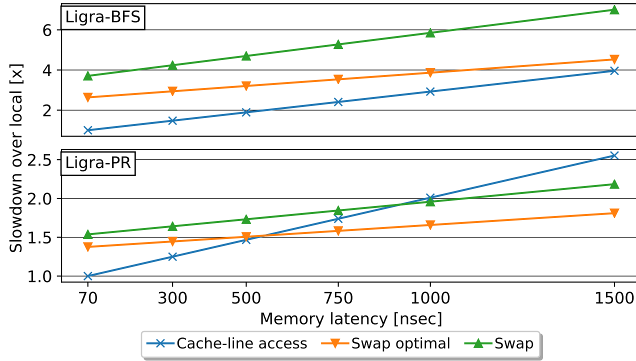


Figure 4. AMAT slowdown over all-local execution for different access models. No single model suits all configurations. Size of local memory: 20% of the working set.

others in all the scenarios, depending on both the application and the system configuration.

In BFS, cache line access performs best across all the evaluated latencies and outperforms even the optimal swap policy. In contrast, cache line access in PR loses its benefits at higher latencies, becoming slower even than the sub-optimal online swap policy.

Takeaway: *Neither the cache line access nor the swap model are superior across applications when we look at the latency regime of disaggregated memory. Therefore, a hybrid design is essential for dynamically determining the correct policy (swap or cache line access) for each accessed memory page in a disaggregated system.*

The most recent state-of-the-art work that utilizes the hybrid model transparently and without dedicated hardware is Nimble [48][2], hence it serves as a baseline in our analysis of current hybrid systems. Other works on disaggregated memory systems that operate in the access latency spectrum of 750–1500 nsec consider only the swap access model [9, 10, 22, 24, 42], which we collectively refer to as "swap" in the rest of the paper.

3.2 Granularity of memory management: base pages vs. huge pages

The use of huge pages (2 MB) is known to lower address translation overheads by extending the TLB reach and reducing the page walk time [30, 38, 39]. Intuitively, huge pages are particularly appealing for systems where an additional memory tier expands the available memory size dramatically (i.e., as more memory can be used by applications).

However, as we show in this section, these benefits diminish when the remote access latency increases, and are eventually outweighed by the negative effects of using large page sizes.

3.2.1 Slow remote memory renders huge pages ineffective.

We run five benchmarks that were shown in prior studies to benefit from using huge pages in local memory [30, 38, 39]. To understand whether these benefits hold in a disaggregated memory system, we pin the working set entirely in remote memory and configure the system to use huge pages. Page tables are placed in local memory for the best performance [6, 29]. We compare huge and base page performance under varying remote memory latencies. This setup is aimed to highlight the performance impact of huge pages where remote memory accesses dominate execution time.

Figure 5a shows diminishing speedups for huge pages as the remote memory latency increases. Huge pages yield up to a 30% speedup when used in a system with local memory only, mainly due to reducing memory access latency by about 70 nsec with faster page walks (recall that a page walk on x86 entails multiple references to the memory hierarchy, whose combined latency may exceed that of fetching the data). However, these savings become negligible when the access latency to remote memory is up to an order of magnitude larger than local memory latency. Then, the latency to fetch the data from remote memory dominates end-to-end latency, resulting in a best-case speedup of 3% when huge pages are used (i.e., page walks are served by local memory, and followed by a data access to remote memory).

Takeaway: *The use of huge pages in remote memory results in small or no performance benefits.*

3.2.2 Hotness fragmentation makes huge pages harmful.

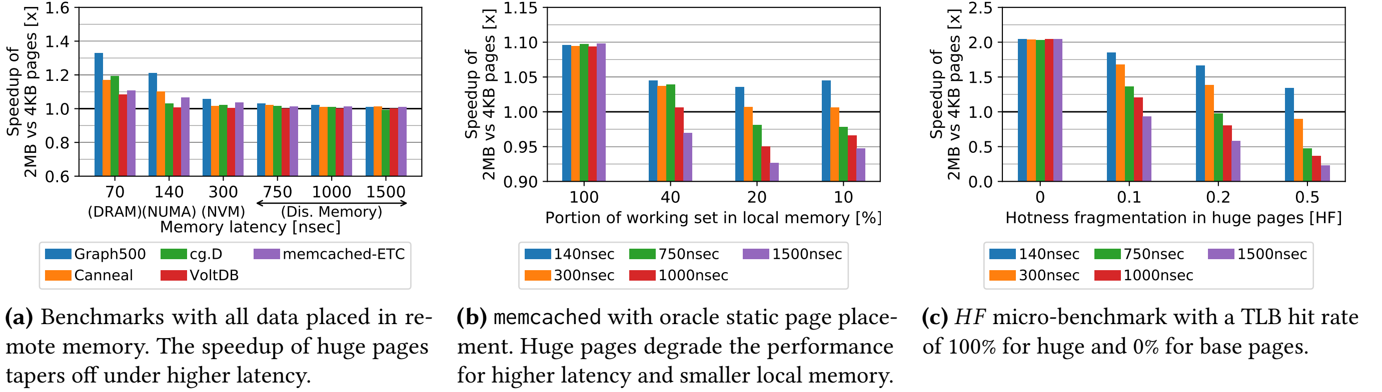
The use of huge pages is often associated with internal huge page fragmentation, also known as *memory bloat* [30, 38]. This phenomenon leads to an increase in memory usage and results in poor memory utilization; i.e., some of the base pages that constitute a huge page are never utilized.

When huge pages are used in a tiered memory system, we observe poor memory utilization that cannot be explained by memory bloat alone. Rather, we find that the *spatial locality of accesses* to huge pages tends to be low. When a huge page spanning the equivalent of 512 base pages is migrated into local memory, its constituent base pages are all accessed but are not equally hot, resulting in sub-optimal use of local memory space. We call this phenomenon *hotness fragmentation (HF)*.

The performance implications of *HF* are particularly acute when the local memory size is a fraction of the working set size and can result in application slowdowns in a disaggregated memory system. We next characterize the effects of *HF* using a representative workload and then provide a more formal analysis.

Intuition: ***HF*** **in `memcached`.** We compare the performance of the `memcached-ETC` workload [21, 31] using base and huge

[2]Nimble was developed for tiered memory systems at a single node where fast DRAM is augmented with slower secondary memory. However, it is equally applicable to tiered memory systems comprised of local and remote DRAM (i.e., disaggregated memory) as stated in the paper.



(a) Benchmarks with all data placed in remote memory. The speedup of huge pages tapers off under higher latency.

(b) `memcached` with oracle static page placement. Huge pages degrade the performance for higher latency and smaller local memory.

(c) *HF* micro-benchmark with a TLB hit rate of 100% for huge and 0% for base pages.

Figure 5. Effect of using huge pages in disaggregated memory

pages. We illustrate the impact of *HF* while using a static policy that places the top hottest pages in local memory, as identified by offline profiling. This policy yields the best-case performance estimate, as it works better than the known best online policy [48] for this workload.

As we reduce the fraction of the working set in local memory, the positive effect of huge pages should level off; i.e., we already saw in Figure 5a that moving the working set to remote memory results in the same performance for base and huge pages (speedup=1). However, due to *HF*, *huge pages in local memory result in lower performance than base pages.* Figure 5b shows that the slowdowns are pronounced for remote memory latencies of 750 nsec and above, the ranges of interest for disaggregated memory systems.

Analysis. We define the measure of hotness fragmentation, *HF*, as $HF = 1 - \frac{N_{huge}}{N_{base}}$, where N_{huge} and N_{base} are the number of accessed base pages in local memory, under huge- and base-page executions, respectively. Low values of *HF* imply that the utility of local memory for huge pages is high. However, high *HF* values correspond to an access pattern with poor spatial locality within huge pages, a pattern accommodated better by using base pages.

To illustrate the performance impact of *HF*, we run a micro-benchmark with a hot working set that fits entirely in local memory when base pages are used. We use a bimodal, uniformly random pattern where the skew of hot pages represents 90% of all accesses. To vary *HF*, we keep the number of hot base pages constant during the experiment but vary the distribution of the accesses across huge pages. Our benchmark is designed to favor huge pages: it ensures all memory accesses result in an LLC miss, has a 100% TLB hit rate when using huge pages, and has a 0% TLB hit rate when using base pages.

Figure 5c shows that with disaggregated memory (remote memory latencies exceeding 750 nsec), *HF* makes huge pages inferior to base pages. With *HF* as low as 0.2 (i.e., 80% of the base pages constituting a huge page are hot), the 2× speedup of huge pages disappears at remote access latency of 750 nsec. At 1000 nsec remote access latency, the execution with huge pages is 20% slower than with base pages.

***HF* in applications.** We now measure *HF* in four real applications: `memcached-ETC` and `Graph500` (where huge pages are advantageous), and `Ligra-BFS` and `Ligra-PR`, which are not sensitive to page size.

We consider two memory placement policies: offline static (top hot pages pinned in local memory) and online dynamic (which migrates pages using Nimble [48]). To compute *HF*, we periodically (every 1 sec) sample and count all the base-page access bits and compute the average over the execution.

The center and bottom rows of Figure 6 show the *HF* statistics for each application, for different sizes of local memory. The top row shows the fraction of base pages accessed in local memory for each policy and page size.

We make several observations. First, as expected, smaller local memory consistently results in higher values of *HF*, for example when 20% of the working set is in local memory, `memcached-ETC` experiences 0.2 and 0.5 *HF* with static and dynamic policies, respectively. In other words, local memory utilization is effectively halved with huge pages under a dynamic policy as compared to the utilization with base pages. These results help explain the performance of `memcached-ETC` in Figure 5b. Second, the dynamic migration policy in the state-of-the-art system [48] causes significant *HF*. Finally, the memory bloat caused by huge pages (blue area in the graph) is relatively small as compared to the *HF*. The results of this experiment indicate that *HF* indeed occurs in real applications, and is correlated to their performance in disaggregated memory systems.

Thermostat [7] considered a related question of the correlation between the huge page's hotness and the number of accesses to base pages in a huge page. They conclude that the spatial frequency of accesses within a huge page is poorly correlated with its true access rate. Their data supports our

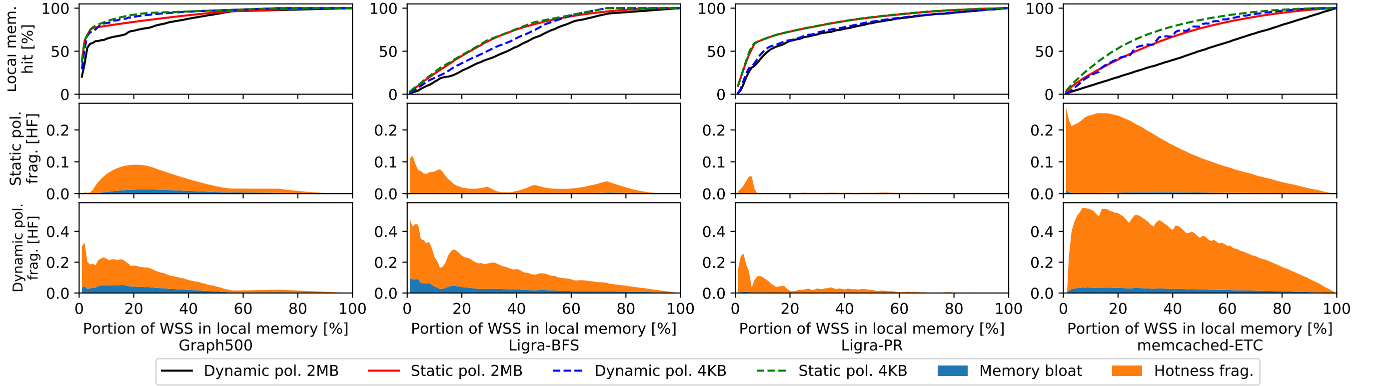


Figure 6. Local memory hit rate (top) and *HF* (center, bottom) vs. fraction of the working set in local memory.

observation that *HF* is a real problem in huge-memory management, but they draw a different conclusion than us that serves their objective: motivating the use of huge pages with low-latency remote memory, where the adverse effects of *HF* is less pronounced.

Takeaway: *The use of huge pages in disaggregated memory systems decreases the effective local memory capacity due to HF. Thus, using them for managing disaggregated memory is detrimental for performance.*

3.3 Granularity of page migration: on-demand vs. batch migrations

Disaggregated memory systems require swift identification and migration of hot pages; when a remote page has been classified as hot, promptly migrating it into local memory reduces the number of high-latency remote memory accesses. Prior work argued that *batch migration*, i.e., grouping multiple pages to perform their migration together, is essential for achieving high migration performance [48]. This is in contrast to *on-demand* page migration such as used in swap-based tiered memory systems [10, 22, 24], which migrate a page in response to a page fault, one page at a time. However, batch migration implies that pages are migrated ahead of time, in anticipation that they will be consequently accessed after the pages have become resident in local memory. In other words, pages are prefetched from remote memory according to the same pages' *prior* access statistics.

Due to this behavior, batch migration might result in imprecise migration decisions (evicting hotter local memory pages), whereas on-demand page-by-page migration is more conservative and costly. This presents a non-trivial trade-off between migration performance and migration accuracy, and prior work argued for optimizing the former at the expense of the latter.

We analyze the costs and benefits of the migration granularity trade-off:

Do batch migrations amortize costs? Prior work [48] observed that migrating a single page via the built-in Linux system call `move_pages` is slow, and propose using batching to amortize such costs. We show that batching is not a prerequisite for high migration performance.

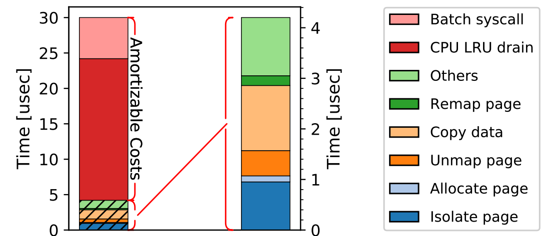


Figure 7. Execution time breakdown of migrating a batch with a single 4 KB page to another NUMA node, using the existing `move_pages` system call.

We start by profiling `move_pages`. As can be seen in Figure 7, 86% (26 μsec) of the time is indeed spent on *amortizable* operations – "batch syscall" and "CPU LRU drain" – performed once per system call regardless of the number of pages that are migrated. But a closer examination reveals that such operations are unnecessary, as explained next.

Before a page can be migrated, it must be removed from the LRU list of its NUMA node, and Linux keeps a small per-core software cache of recent LRU list entries to reduce contention accessing the global LRU lists. The system call `move_pages` is designed to "drain" (i.e., flush) all per-core LRU list caches to their corresponding global LRU list before migrating any page. This operation is expensive and is conducted via inter-processor interrupts (IPIs) to all cores in the system.

However, we argue that draining is not essential, as pages in LRU caches can temporarily be considered non-migratable. The LRU caches are only 16 entries, constituting only a small fraction of the total memory and leaving many other potential candidates for migration; even a large multi-core system would only have a few hundreds of such pages. Furthermore, LRU caches are constantly updated, such that the desired page may soon leave the LRU cache and its migration would

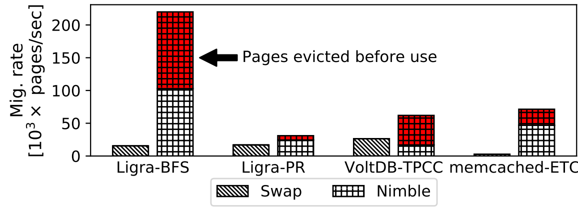


Figure 8. Page migration rate in the on-demand migration (aka swap) and batched migrations (aka Nimble with base pages [48]). Local memory size is 20% of the working set. Batching causes aggressive migrations with low good-put.

be re-enabled. In fact, the *autoNUMA* subsystem in Linux already implements this optimization.

We conclude that per-page migration time can be reduced to 4 μsec (from 30 μsec) that is *not amortizable*: the remaining operations must be performed for every page in a batch.

Takeaway: *Batching is not required for high-performance page migration.*

Do batch migrations bring the correct pages? The effectiveness, or *good-put*, of the batch migration is critical in a system with limited local memory. We define good-put as the proportion of pages that have been migrated to local memory and accessed *at least once* before being evicted.

We measure the good-put of four memory-intensive benchmarks (see § 6 for details) on the Nimble system with batch migrations of base pages. Figure 8 shows that Nimble's good-put is low in three out of the four benchmarks, i.e., 27% in `VoltDB-TPCC` and about 46% in `Ligra-BFS`.

We also observe that the average migration rate of Nimble is considerably higher than on-demand page migration.

Takeaway: *Batch migration leads to a lower good-put and higher migration bandwidth than on-demand migration.*

4 HotBox

Based on the insights of the analysis in § 3, we design and implement HotBox, a novel memory management subsystem for disaggregated memory systems with four design goals:

1. Use of base pages to manage memory: to avoid *HF*,
2. On-demand hot-page migration under the hybrid access model: for improved migration utility,
3. Adaptive memory scanning: for improved accuracy of eviction decisions under high local memory pressure while retaining low overheads in a common case,
4. Adaptive throttling of migration mechanisms according to the utility of past migrations: to avoid local memory thrashing.

The first two goals are derived from our analysis (§ 3), which argues for finer-granularity page management (single base page), whereas the last two aim to reduce the inherently higher overheads of fine-granularity page management. We explain the rationale for items 3 and 4 here and provide technical details about the complete system in Section 5.

4.1 Achieving high accuracy of migration decisions with low overheads

Similar to prior tiered memory proposals [7, 26, 48], HotBox uses the access bits of pages in local and remote memory to estimate page hotness. However, we observe that the rate at which the access bits are scanned (i.e., *scanning rate)* is a crucial parameter for correctly identifying migration candidates both on the remote side (to migrate in) and on the local side (to evict out). Intuitively, a higher scanning rate enables more accurate differentiation between frequently accessed pages. That is, the higher the rate, the more frequently a page should be accessed in order to be considered hot. Lower scanning rates naturally result in lower accuracy of the hotness statistics as many pages may end up being marked hot despite a potentially large difference in their access frequency.

While the observations above hold true both for local and remote memory, we find that local memory is particularly sensitive to the scanning rate since it serves as a cache for the remote tier and the majority of the pages are accessed relatively frequently. As a result, the ability to discriminate between hottest and less hot pages is crucial in order to make well-informed eviction decisions [40]. Problematically, a high scanning rate is a costly proposition in terms of CPU cycles, especially when utilizing base pages, as more scanning operations are required to traverse a range of memory when compared to huge pages. While potentially acceptable if used for short intervals of time for local memory, frequent scanning incurs unacceptably-high overheads given large capacities of disaggregated memory.

What is needed is a mechanism that can dynamically navigate the cost-accuracy curve based on workload behavior in regard to memory pressure. To that end, HotBox employs two independent scanners for local and remote memory that interact via a closed feedback loop: (1) remote memory is scanned at a fixed rate that yields low overheads, and (2) local memory is scanned at an adaptive rate tied to the local memory pressure. Only when a remote page is classified as hot, it is migrated to local memory, and will subsequently increase the local memory pressure, resulting in a higher scanning rate. Thus HotBox ensures that the most frequently accessed pages in local memory are correctly identified as hot and are not evicted.

4.2 Avoiding useless migrations

As mentioned above, scanning of access bits must necessarily occur more frequently (thus incurring a higher overhead) to better discern page hotness and improve the hit rate in local memory. However, when the hot working set size is too large, migrations stop being useful (since local and remote working sets have similar hotness) and may cause thrashing of local memory, hence hurting performance. To demonstrate this effect, we measure the throughput of `memcached` serving a

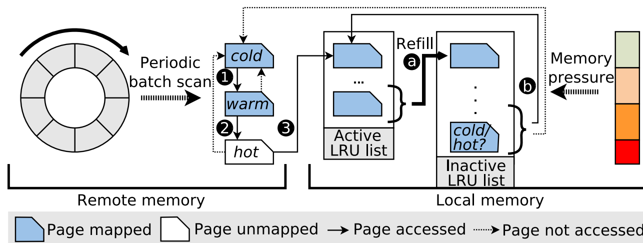


Figure 9. Overview of local and remote page scanning and migration in HotBox.

random (non-skewed) set of requests [31] using the Nimble page migration mechanism. We then run the same experiment, but *disable* the migrations after a warmup and measure the throughput again. We observe a 2.1× higher throughput with migrations disabled. This improvement is expected as the access pattern has no spatial locality, effectively causing local memory thrashing and useless page migration.

To counteract such cases, we devise a simple mechanism to evaluate the *migration utility* and pause migrations when the utility is low. Specifically, we monitor the application's accesses to local and remote memory, which allows the momentary local memory hit rate to be estimated. If this value remains unchanged over several measurement intervals, it indicates that migrations are not helpful in reducing accesses to remote memory and migration is paused. If the value deviates from the average by a few percentage points, migration is resumed. The intuition is that a change in either direction implies a change in the application access pattern. Stopping migrations averts the overheads of local and remote memory scanning and page migrations in cases where the system is in a steady-state and would otherwise continue to cycle pages fruitlessly between the local and remote memory.

5 Implementation

Favoring on-demand page migration reduces the implementation complexity significantly, allowing us to repurpose the existing *autoNUMA* and swap subsystems in Linux. HotBox-introduces only minor modifications into the kernel (less than 2K LOC) and yet meets our design goals.

At a high level, we use (1) the swap subsystem to track and evict cold local pages, modified to use remote memory as the destination for evictions and (2) the autoNUMA subsystem to implement on-demand page migrations, modified to select only hot pages in remote memory and trigger evictions (via the `kswapd` daemon) if there is local memory pressure.

We now briefly survey the existing mechanisms in Linux that we reuse in HotBox, and then, we highlight the modifications necessary to adjust these mechanisms to our needs.

5.1 Existing Linux Subsystems used in HotBox

Page and task balancing with *autoNUMA*. The *autoNUMA* subsystem co-locates processes and the pages they use to minimize memory access latencies [18] in a NUMA system. autoNUMA periodically unmaps a subset of pages in a process, accesses to which triggers a page fault, which the kernel uses to keep track of the current NUMA node of the faulting page and process. With this information, autoNUMA reschedules tasks to the NUMA node where most of its accessed pages reside, while at the same time it migrates repeatedly accessed pages into the process's NUMA node.

Page reclamation with `kswapd` and direct reclaim. Linux uses a *page reclamation* algorithm that swaps cold pages out from memory into a swap device. When the system detects memory pressure during the allocation of new pages, it wakes the `kswapd` daemon to free some additional pages asynchronously. If no pages are free, the kernel uses the *direct reclaim* path to free a small batch of pages synchronously.

When a new page cannot be allocated, direct reclaim is invoked. If a page allocation request can be fulfilled, but the resulting number of free pages in the system is lower than a configurable threshold (i.e., when there is memory pressure), the *low watermark*, the `kswapd` daemon is woken. The daemon attempts to relieve memory pressure by freeing multiple pages until a sufficient number are available, the configurable *high watermark*. Both subsystems gradually increase the aggressiveness of their heuristics when a sufficient number of pages cannot be freed. For example, direct reclaim will free dirty pages only when it is most aggressive since they require I/O. Instead, `kswapd` frees a skewed amount of file-backed and anonymous pages that change with the level of its aggressiveness.

To estimate page utility, the page reclamation algorithm classifies pages into two separate LRU lists (active and inactive). It has two sets of lists, for file-backed and anonymous pages, respectively. It also keeps track of two per-page bits: accessed (in the page table), and referenced (in the kernel's per-page metadata). The algorithm iteratively attempts to reclaim pages, becoming more aggressive at each iteration, until it reaches its target. At each iteration, it moves the bottom of the active LRU lists into the top of the inactive LRU lists, until their sizes conform to a preconfigured ratio. Then, it starts scanning the bottom of the inactive LRU lists until it reaches its target and, for each page, checks the accessed and referenced bits to determine whether the page is to be reclaimed or recirculated in the LRU lists [14].

All pages eventually end up at the tail of the inactive LRU list, giving the system an opportunity to scan them. The reclamation algorithm thus constantly recirculates the hottest pages and swaps the remaining pages out. In practice, the estimated utility of a page changes according to memory pressure. The greater the memory pressure, the faster the system cycles through the pages it scans, effectively increasing the average per-page scanning frequency.

Page allocation. When allocating a new page, the kernel first attempts to use the NUMA node in which it is executing.

If this fails, it attempts to use other NUMA nodes until it finds a free page (or none is left in the system).

5.2 Extensions to existing Linux subsystems

We implement HotBox in Linux kernel version 4.9.99, tying the independent production-hardened heuristics present in Linux with new mechanisms that synergistically work to meet our goals.

1) We expose local and remote memories as separate NUMA nodes in the kernel.

2) We implement a new remote memory scanning algorithm that performs *on-demand* migrations of hot pages to increase migration utility. When enabled by the migration utility monitor, the algorithm, shown in Figure 9, scans blocks of remote pages periodically to check their access bits (with a configurable batch size and period for scanning, effectively capping the migration bandwidth).

To further reduce the scanning overheads of remote memory, we perform a hierarchical scan of the page table radix tree and sample the accessed bits of the higher-level page tables (`PMD`): if the accessed bit is `0`, we skip scanning all the entries in the underlying page table (`PTE`), as no page inside the page table has been accessed [25].

Remote memory pages' hotness is tracked with a per-page state that is updated by the periodic scans (❶ cold, ❷ warm, and ❸ hot), and hot pages are migrated into local memory on-demand (after being unmapped and subsequently accessed; if no local pages are free, the page is re-mapped).

3) We modify `kswapd` to evict local cold pages into remote memory. When triggered, the modified `kswapd` refills and scans the inactive LRU list as usual (Steps ⓐ and ⓑ, respectively, in Figure 9). If a cold page is found, it is evicted.

4) We modify page allocation to ensure kernel pages and all page tables are placed in local memory (for performance [6]), while other pages have a "best effort" placement, as in Linux.

5) We modify page allocations to *not* trigger direct reclaim when attempting to migrate a remote page, since an aggressive direct reclaim can stall applications for long periods of time and it is frequently better to keep the page remotely.

5.3 Monitoring migration utility

We monitor the migration utility by maintaining a windowed average of the ratio between local and remote memory accesses of each task in the system (using the `UOPS_LLC_MISS_RETIRED.{REMOTE,LOCAL}_DRAM` PMU counters [25] via the in-kernel `perf_event` API) and consider the system to be in a steady state when the ratio has remained stable across three time windows.

6 Evaluation

We aim to answer two key questions: is HotBox effective in running applications in disaggregated memory with realistic access latency, and how does it compare with the state-of-the-art tiered-memory OS management systems?

Processor	2× Intel E5-4620 v2 w/ PMEP microcode patch
Memory	2× 64 GB DDR3 1333 Hz
Network	Intel 10-Gigabit SFI
OS	Ubuntu 16.04, Linux version 4.9.99

Table 2. PMEP platform.

6.1 Methodology

As system support for cache-line remote-memory accesses is not commercially available as of yet, we obtained the results in this paper using Intel's Persistent Hybrid Memory Emulation Platform (PMEP [19], Table 2), which was also used for studying tiered memory in prior research [20]. PMEP runs Linux, exposes a remote memory tier as a separate NUMA node, and uses special CPU microcode to emulate slower memory access latencies by injecting additional stalls, thereby allowing end-to-end full-system performance measurements. We verified that PMEP injects the requested latencies using Intel's MLC tool [46].

All applications run on the first NUMA node. We disable HyperThreading and all unused CPUs. Local memory is placed on the first NUMA node. To limit its size, we use a simple balloon process that allocates and locks pages in memory. Certain workloads require a client application over the network, in which case we use an identical PMEP machine without HotBox connected via 10 Gbps Ethernet.

6.2 System and benchmark configurations

We evaluate HotBox against four alternative approaches:

All-local: Data pages are located in local memory; no migration. This serves as an upper bound on performance.

All-remote: Data pages are located in remote memory; no migration. This serves as a lower bound on performance.

Swap: Remote memory is handled as a swap device; pages are paged-in and out via the Linux LRU mechanism. This is used to illustrate the performance of systems that do not perform direct access to remote memory (not hybrid) [10, 22, 24]. This is an optimized implementation that eliminates a block device layer similarly to `frontswap` [1]. It is 1.4× faster than Linux swap when mounted on BRD [5].

Nimble [48]: Nimble migration management is evaluated using the same number of CPU cores as HotBox. In this configuration, their concurrent page migration mechanism degrades performance, and hence, we disable it. We note that Nimble's technique that enforces the local memory limit does not count kernel allocations (e.g., page tables) and file-mapped pages (e.g., code) as part of local memory consumption, giving it a slight advantage over HotBox.

We evaluate Nimble with both base and huge pages and the other approaches with base pages. In addition, we configure

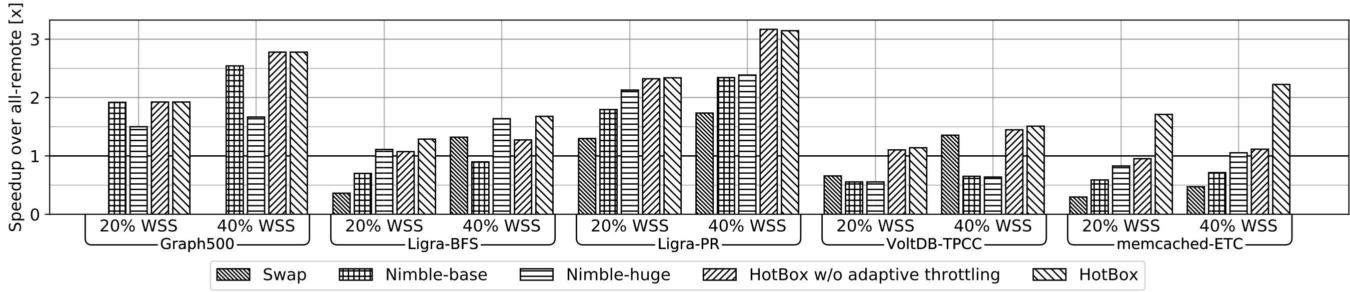


Figure 10. End-to-end speedup over all-remote execution for different workloads and local memory sizes (WSS – working set size). Higher is better. Remote memory latency is 750 nsec. "Swap" did not finish for `Graph500`.

the maximum migration rate to be 2 GB/s in both Nimble and HotBox. This value provides an empirically good performance across all configurations for Nimble.

We run five benchmarks with a working set of 10 GB each:

`Graph500`: A known graph processing benchmark [35] with two kernels: graph construction and BFS. It was used in a prior study [48]. The access pattern is random.

`Ligra-BFS`: BFS on a popular `Ligra` graph processing framework [44]. We use the included `rMatGraph` utility to create a graph with 40 M vertices and 400 M edges with the parameters from `Graph500` [35] (A=0.57, B=C=0.19). We exclude graph loading time. The access pattern is random.

`Ligra-PR`: Page Rank (PageRankDelta) on `Ligra` (same input as in BFS). The access pattern is initially random, but the algorithm reduces the effective working set over iterations.

`VoltDB-TPCC`: TPC-C benchmark (we use 256 warehouses and 4 sites) [17] running on `VoltDB` [45], an in-memory, row-based relational database. We measure the average transaction latency over 5 min. The access pattern is random, but accesses are in blocks of 8 KB (the record size); there are also many random accesses to a small number of pages with data structures used by the Java Virtual Machine.

`memcached-ETC`: An in-memory KV store [21] measured using the Mutilate client [31] and Facebook's ETC benchmark [11]. The throughput is measured after loading 30 M records and a 1 min warmup. We average the throughput over 5 min. The access pattern is random with a skew; 90% of requests account for 10% of the keys. Despite this skewness, the distribution of the popular keys in *memory* is random.

All the benchmarks use four threads pinned to different CPU cores, except for `memcached`, which runs in a single thread (we do not have enough machines to saturate more).

6.3 End-to-end evaluation

We start by evaluating the end-to-end performance of HotBox, compared to the state-of-the-art in tiered memory systems. We also evaluate HotBox without the adaptive throttling mechanism. In this study, we model a remote memory access latency of 750 nsec, which is optimistic for near- to mid-term time frame; other latencies are explored below. We evaluate the workloads with different local memory sizes that correspond to a fraction of the working set size (WSS) of the application; e.g., a point labeled "20% WSS" indicates a local memory size that can fit 20% of the application's working set. We thus focus on relative trends to decouple our evaluation from the constraints imposed by fixed choices of dataset and memory sizes.

Figure 10 shows that HotBox performs best on all studied configurations. The benefits of HotBox are particularly high for `Ligra-PR`, `memcached-ETC` and `VoltDB-TPCC`, with a relative speedup of up to 2.25× over the best performing state-of-the-art configuration. These workloads have a skewed memory access pattern where HotBox can provide more effective identification and migration of hot pages (§ 6.5 further analyzes the effectiveness of the migration decisions).

`Graph500` exemplifies our hypothesis that *HF* and huge pages together lead to bad performance. This benchmark has large huge page speedups at low memory latencies (above 1.3× in Figure 5a), but also has *HF* of 0.13 to 0.2 for the 20-40% WSS figures (see Figure 6). As a result, the huge page optimizations in Nimble-huge are inefficient at higher remote memory latencies, and the same system with base pages, Nimble-base, performs 1.2×-1.6× better for 20%-40% WSS (as explained in § 3).

`Ligra-BFS` and `memcached-ETC` show that HotBox's adaptive mechanisms for avoiding page thrashing are also critical for performance. HotBox without adaptive throttling exhibits noticeably lower performance compared to HotBox in these workloads, demonstrating the adaptive throttling mechanism's benefits. We also note that Nimble-huge yields higher performance than Nimble-base in these workloads as a result of lower page management overheads. These are workloads where we know the hottest pages do not fit in local memory and, therefore, all policies but HotBox continuously thrash the local memory with migrations.

Interestingly, `VoltDB-TPCC` at 40% WSS shows that Swap outperforms Nimble by 2.3× and is almost as good as HotBox. This benchmark has sequential accesses to 8 KB blocks, and we believe that the read-ahead prefetcher in the Linux swap cache is identifying this pattern and reducing on-demand

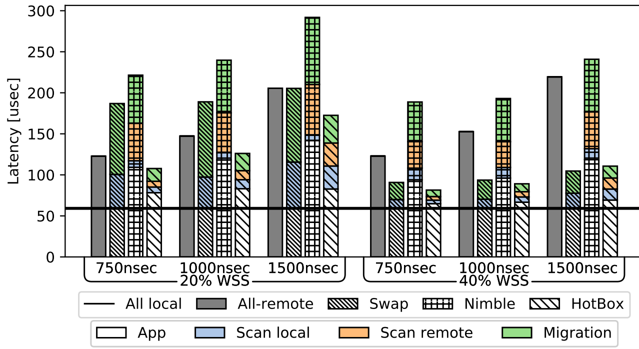


Figure 11. Runtime breakdown of `VoltDB-TPCC`. Lower is better.

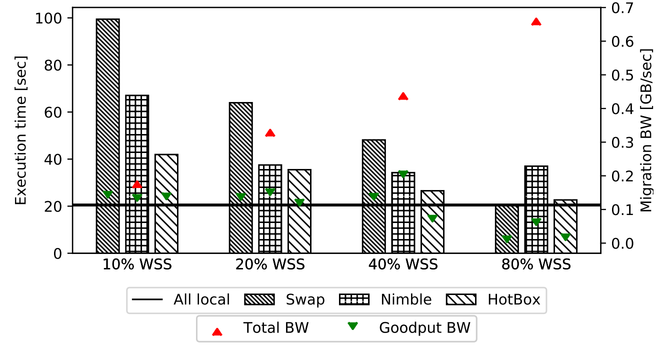


Figure 12. Runtime and migration bandwidth of `Ligra-PR`. Lower is better.

page migrations. This shows prefetching policies could have a secondary but important role, as HotBox still outperforms all policies without using prefetching.

In summary, *HotBox outperforms all other systems by challenging traditional tiered memory management designs*. This does not mean that previous memory management optimization efforts such as prefetching are unnecessary, but rather, as we will see next, they become secondary to local memory hit rate in the latency ranges of disaggregated memory.

6.4 Latency sensitivity and system overheads

Here we study the balance between system overheads and the accuracy of migration decisions on both Nimble and HotBox. To this end, Figure 11 shows absolute performance numbers and a breakdown of the system overheads for `VoltDB-TPCC` under different memory latencies. Nimble is invoked with base pages to isolate the impact of batching (Figure 10 above already showed that huge pages rarely provide a substantial benefit, sometimes having severe negative impacts).

The breakdown shows the runtime of the user application ("App"), the time spent scanning local and remote memory access bits ("Scan local" and "Scan remote," respectively), and the time spent migrating pages ("Migration"). We chose `VoltDB-TPCC` as it exhibits a random but skewed access pattern where the hottest pages fit local memory at 40% WSS.

First, recall that Nimble performs aggressive migration with relatively low good-put for this workload (§ 3, Figure 8). The implications of this are evident here: migration overheads are much higher than in HotBox, and the fraction of the application time is larger and increases with the remote memory latency, indicating a significant amount of remote memory accesses. This is not the case for HotBox, which has better migration decisions (lower "App" time, in some cases close to optimal performance). The lower "App" time also indicates that with an additional dedicated core to offload both mechanisms' overheads ("Scan" and "Migration"), HotBox will still exceed Nimble's performance.

Furthermore, HotBox is more resource efficient: under 40% WSS, HotBox's performance without an additional core (including the "Scan" and "Migration" overheads) still outperforms Nimble's "App" time alone, assuming an additional core is able to hide Nimble's overheads.

HotBox increases the memory scanning rate as a function of memory pressure, which is reflected in the "Scan local" and "Migration" times. These times are higher for 20% than for 40% WSS. Thus, this design enables gradual and more accurate migration decisions, resulting in smaller "Migration" and "App" overheads due to fewer and more accurate migrations.

To conclude, the overheads of *HotBox are lower while, at the same time, its adaptive local scanning rate design provides better performance even at high remote memory latencies.*

6.5 Effects of local memory size on migration bandwidth

Local memory size is decisive in application performance. Figure 12 shows the execution time and effective migration bandwidth consumed by the system under different local memory sizes. Again, Nimble is executed with base pages. HotBox effectively executes with aggressive local memory size reductions (e.g., 10% WSS). Starting at 40% WSS, HotBox can effectively identify and migrate-in all hot pages. At 80% WSS, the migration bandwidth is close to zero, similar to that of Swap, and both are on par with the all-local execution. This implies that the hot working set of `Ligra-PR` represents approximately 80% of its full data set.

Notably, the migration bandwidth of Swap and HotBox decreases with a larger local memory, but Nimble's total bandwidth increases almost 5× as compared to its values with a small local memory. This is due to Nimble's page-exchange batching mechanism: Nimble's migration batch size is dependent on the size of the local memory; the batch size is the maximum between the number of cold pages in local memory and the hot pages in remote memory. As the local memory size increases, more victim pages are gathered

from local memory, enabling larger batches with low goodput ratios.

In conclusion, *as compared to all the other approaches, HotBox is more effective at a reduced local memory size.*

7 Related work

Tiered memory systems have attracted considerable research attention, with many innovative ideas being proposed across both hardware and software stacks [7, 9, 10, 15, 20, 24, 26, 28, 32, 36, 37, 42, 48]. Here, we review the most relevant research work in the context of OS support for memory disaggregation.

Swap-only access models. Gao et al. analyzed the relationship between network and application performance [22], using a swap device implementation that injects additional latencies. Infiniswap [24] applies a similar mechanism to utilize excess memory in remote servers. LegoOS [42] proposes a disaggregation design, where local memory acts as a coarse-grain cache of remote memory with the help of specialized hardware support. These and other approaches [9, 10] differ from HotBox in that they do not support direct cache-line access to remote memory, and hence, their migration mechanisms and policies differ from ours. Further, their authors did not analyze the impact of huge pages, which is one of our core contributions.

Leap [9] presents an efficient page prefetching from remote memory in the swap model. HotBox can benefit from such a prefetching policy: prefetching policies for the hybrid model can gain at least the same performance benefits as policies in the swap model (since the hybrid model includes the swap model). Prefetching for the hybrid model presents additional challenges, such as evaluating remote memory access patterns under direct cache-line accesses with low overhead, that we leave for future work.

Hybrid access models. The approach most relevant to ours is the Nimble page management system [48], which uses huge pages and batch migrations to accelerate page migration in a tiered memory system with 200 nsec remote memory latency. In contrast, our analysis highlights the negative side effects of these optimizations, in particular for higher-latency disaggregated memory, leading to different design decisions in HotBox, which result in better performance.

Similarly, Thermostat [7] targets tiered memory with the NVM latency range, focusing on the differentiation and placement of hot and cold huge pages. We demonstrate that this can hamper performance in a disaggregated memory setting.

HeteroOS [28] targets virtualized systems and provides a guest-hypervisor cooperation mechanism for page placement in tiered memory systems. Our use of performance counters to reduce the migration overheads is similar to theirs. Our huge-page analysis is complementary to HeteroOS, as it provides the necessary first step toward investigating the performance tradeoffs of utilizing huge pages in virtualized environments.

Non-transparent access models. Dulloor et al. [20] introduced a toolset to guide application data placement on a tiered memory system using profile data. HotBox employs dynamic memory management with online decisions, and hence, it is not directly comparable. Other works [41, 47] rely on application data and require code changes to support disaggregated memory. HotBox runs at the kernel level and is applicable to all applications transparently.

Hardware-based disaggregated architectures. A number of proposals considered new hardware support for disaggregated memory [15, 32, 37, 42]. In particular, Scale-out NUMA [37], proposes a new hardware architecture for a non-coherent distributed compute and memory system with direct remote access capabilities, laying the foundations for future byte-addressable disaggregated systems.

Huge pages. Prior studies indicate that the use of huge pages may hinder performance in a NUMA system due to false sharing of pages between NUMA nodes [23] and may occupy additional memory resources due to memory bloat [30, 38, 39]. Our research is complementary, as we identify a new phenomenon, hotness fragmentation, that affects performance specifically in disaggregated memory systems.

8 Conclusions

Memory disaggregation offers promising operational savings in a data center to meet the ever-increasing application memory requirements. Existing OS approaches for managing tiered memory [7, 20, 26, 28, 48] target memory latencies lower than those offered by current and near-future disaggregated memory latencies [2–4, 13, 16].

In this paper we show that the existing memory management approaches are ill-suited in the latency regimes of disaggregated memory, and introduce HotBox, a novel memory management subsystem for the Linux kernel targeting disaggregated memory. Based on a detailed analysis of state-of-the-art work and conventional memory management techniques, HotBox follows four design choices: to use only base pages for migration, to increase eviction accuracy by sampling page hotness according to memory pressure, to migrate remote memory pages only upon access to increase migration utility, and to throttle migration and page hotness sampling according to the utility of past migrations. These design choices enable HotBox to outperform state-of-the-art systems by up to 2.5× on memory-intensive workloads. HotBox carefully reuses and extends existing Linux kernel subsystems, minimizing the impact on the existing kernel code (less than 2K LOC) and simplifying its integration in future releases.

Acknowledgments

We thank our anonymous reviewers for their insightful comments. This work was partially supported by the Technion Hiroshi Fujiwara Cyber Security Research Center, the Israel National Cyber Directorate, and the Israel Innovation Authority. We gratefully acknowledge support from Israel Science Foundation (Grant 1027/18).

References

[1] 2015. Frontswap. https://lwn.net/Articles/386103/.

[2] 2018. *Gen-Z Core Specification 1.0.*

[3] 2019. *Compute Express Link Specification.*

[4] 2021. Connectx-6 single/dual-port adapter supporting 200Gb/s with VPI. https://www.mellanox.com/page/products_dyn?product_family=265&mtag=connectx_6_vpi_card,.

[5] 2021. Linux Block Ram Disk. https://www.kernel.org/doc/html/latest/admin-guide/blockdev/ramdisk.html.

[6] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* 283–300.

[7] Neha Agarwal and Thomas F Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 631–644.

[8] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing.* 121–127.

[9] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20).* 843–857.

[10] 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.

[11] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40. ACM, 53–64.

[12] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966), 78–101.

[13] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. 2015. Intel® Omni-path architecture: Enabling scalable, high performance fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects.* IEEE, 1–9.

[14] Daniel P Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.".

[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] CCIX Consortium 2019. *CCIX Base Specification Revision 1.1. Version 1.0.* CCIX Consortium.

[17] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. 2011. TPC-E vs. TPC-C: characterizing the new TPC-E benchmark via an I/O comparison study. *ACM SIGMOD Record* 39, 3 (2011), 5–10.

[18] Jonathan Corbet. 2012. AutoNUMA: the other approach to NUMA scheduling. *LWN. net* (2012).

[19] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems.* ACM, 15.

[20] 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.* ACM, 15.

[21] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.

[22] 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.

[23] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Fuston, Alexandra Fedorova, and Vivien Quéma. 2014. Large pages may be harmful on NUMA systems. (2014).

[24] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient Memory Disaggregation with Infiniswap.. In *NSDI.* 649–667.

[25] Part Guide. 2011. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part* 2, 11 (2011).

[26] Vishal Gupta, Min Lee, and Karsten Schwan. 2015. Heterovisor: Exploiting resource heterogeneity to enhance the elasticity of cloud platforms. *ACM SIGPLAN Notices* 50, 7 (2015), 79–92.

[27] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[28] 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.

[29] Sandeep Kumar, Aravinda Prasad, Smruti R Sarangi, and Sreenivas Subramoney. 2021. Radiant: efficient page table management for tiered memory systems. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management.* 66–79.

[30] 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.

[31] Jacob Leverich. 2014. Mutilate: high-performance memcached load generator.

[32] Yang Li, Saugata Ghose, Jongmoo Choi, Jin Sun, Hui Wang, and Onur Mutlu. 2017. Utility-based hybrid memory management. In *2017 IEEE International Conference on Cluster Computing (CLUSTER).* IEEE, 152–165.

[33] 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. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 267–278.

[34] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[35] 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.

[36] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. 2018. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference*. 1–12.

[37] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. *ACM SIGPLAN Notices* 49, 4 (2014), 3–18.

[38] Ashish Panwar, Sorav Bansal, and K Gopinath. 2019. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 347–360.

[39] Ashish Panwar, Aravinda Prasad, and K Gopinath. 2018. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 679–692.

[40] John T Robinson and Murthy V Devarakonda. 1990. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. 134–142.

[41] 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 20)*. 315–332.

[42] 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 18)*. 69–87.

[43] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A network architecture for disaggregated racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 255–270.

[44] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.

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

[46] V Viswanathan, Karthik Kumar, and T Willhalm. 2013. Intel memory latency checker. *Intel Corporation* (2013).

[47] 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 20)*. 261–280.

[48] 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*. ACM, 331–345.

[49] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 169–182.