PIPM: Partial and Incremental Page Migration for Multi-host CXL Disaggregated Shared Memory

Gangqi Huang
University of California, Santa Cruz
Santa Cruz, California, USA
alexhuang1403@ucsc.edu

Heiner Litz
University of California, Santa Cruz
Santa Cruz, California, USA
hlitz@ucsc.edu

Yuanchao Xu
University of California, Santa Cruz
Santa Cruz, California, USA
yxu314@ucsc.edu

Abstract

The emerging Compute Express Link (CXL) interconnect supports multi-host cache-coherent disaggregated shared memory (CXL-DSM). However, existing page migration approaches, designed primarily for single-host systems, are inefficient in multi-host CXL-DSM scenarios. To address this, we propose Partial and Incremental Page Migration (PIPM), a hardware-based solution that transparently leverages host-side local memory. PIPM is co-designed with the CXL multi-host coherence protocol, enabling coherent access to data residing in local DRAM. To overcome limitations of existing migration methods, PIPM supports fine-grained data migration and integrates hardware-based monitoring and decision-making mechanisms to optimize data placement. Evaluation results demonstrate that PIPM delivers performance improvements of up to 2.54× (1.86× on average) over the default multi-host CXL-DSM configuration.

Keywords: Distributed Shared Memory, Disaggregated Memory, Page Migration, Cache Coherency, Compute eXpress Link

1 Introduction

Emerging applications in AI [1], databases [2], and big-data analytics [3] increasingly demand higher memory capacity, greater bandwidth, and lower costs [4–23]. With the slowdown of DRAM technology scaling [24, 25], architects have turned to Compute Express Link (CXL) for flexible, disaggregated shared memory (CXL-DSM), significantly improving efficiency and reducing DRAM costs [11, 25–27]. The latest CXL standards (CXL 3.x) further support coherent multi-host shared memory, enabling dynamic compute resource allocation and flexible memory partitioning, enhancing throughput and cost efficiency [2, 27–30].

Recent research highlights substantial benefits of multi-host CXL-DSM across various applications [2, 28, 30–35]. For example, HydraRPC uses CXL-DSM to improve RPC scalability [28], CXLfork reduces local memory consumption by 87% on average for cross-host process cloning [33], Tigon achieves an average 2.5× throughput improvement for databases compared to configurations without CXL-DSM [2], and PolarCXLMem [30] shows an up to 154.4% performance improvement compared to RDMA-based cloud databases [29, 36, 37].

Despite its potential, CXL-based system performance is often limited by the high latency of remote CXL memory accesses [11, 25, 30], which are typically two to three times slower than local DRAM accesses upon LLC misses [25]. A common solution is page migration [11–14, 38–41]: pages identified as frequently accessed by a host are migrated from CXL memory to a host's local memory, converting subsequent remote accesses into low-latency local accesses.

However, existing page migration schemes designed for single-host CXL disaggregated memory are ineffective in multi-host CXL-DSM for two reasons: **(1) Local gain, global pain.** In single-host systems, migrating a hot page to local DRAM is strictly beneficial, assuming sufficient local memory capacity is available. In a multi-host CXL-DSM, however, moving a hot page from shared CXL memory to one host's local DRAM may harm overall performance, outweighing the local benefit. To preserve coherence and consistency, the migrated page needs to become non-cacheable for all other hosts. As a result, remote accesses incur extra hops, round-trips, and address remapping overheads, significantly increasing latency for other hosts accessing the page. **(2) Poor Migration Scalability.** The side-effects of page migration in multi-host CXL-DSM systems pose significant challenges for supporting efficient and timely migration. However, migration overheads grow significantly as page migration is no longer entirely local but instead requires coordination across hosts, including CXL RPCs [28], per-host page-table updates and TLB shootdowns.

To address these challenges, we propose **Partial and Incremental Page Migration (PIPM)** for multi-host CXL-DSM. **Partial Migration:** Instead of migrating entire pages into local memory or retaining them fully in CXL memory, PIPM selectively migrates only those cache blocks frequently accessed by a host into its local memory, while keeping less-frequently or remotely accessed blocks in CXL memory. This selective strategy differentiates local from inter-host access patterns at a fine granularity, effectively resolving the "local-gain, global-pain" issue. Moreover, by maintaining two possible destinations for cache blocks, PIPM significantly reduces migration management overheads, such as page-table updates and TLB invalidations. **Incremental Migration:** Rather than explicitly migrating entire pages, which incurs substantial data-transfer overhead, PIPM leverages intrinsic memory accesses of programs to migrate cache blocks incrementally and selectively. Specifically, PIPM determines

whether to incrementally migrate cache blocks from CXL memory into the requester host's local memory or back to CXL memory during cache coherence request handling. Consequently, incremental migration involves no additional data transfers beyond regular cache-fill and eviction operations. The partial migration policy identifies cache blocks and target hosts without initiating immediate data transfers, relying entirely on incremental migration for data movement. Together, these techniques enable PIPM to systematically address the previously identified challenges.

We develop architectural support for PIPM, including a majority-vote migration policy, a two-level hardware remapping table, and PIPM-coherency to effectively enable partial and incremental page migration. We evaluate our technique using the Championship simulator [42, 43]. PIPM achieves an average speedup of 1.86× on multi-host CXL-DSM systems and surpasses six state-of-the-art methods.

Overall, this paper makes the following contributions:

1. Qualitatively and quantitatively identifies the challenges of page migration in multi-host CXL-DSM.
2. Introduces partial and incremental page migration to systematically address these challenges.
3. Presents an architectural design that effectively and efficiently supports partial and incremental page migration.
4. Provides a comprehensive evaluation demonstrating the effectiveness of PIPM.

2 Background

2.1 CXL Disaggregated Shared Memory

The CXL 3.0 standard introduces CXL Disaggregated Shared Memory (CXL-DSM) [26, 44–49], allowing a pool of CXL memory to be coherently shared across multiple hosts. This contrasts to prior versions of CXL in which the CXL pool had to be statically partitioned and each partition assigned to one particular host. Also note that CXL 3.0 only allows coherent sharing of the CXL memory pool, while each host's local memory remains invisible to other hosts.

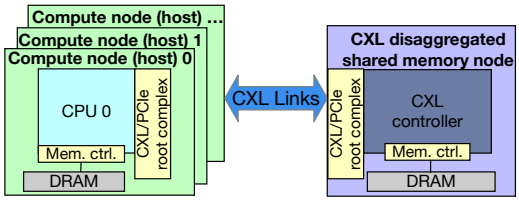

Figure 1. Multi-host CXL-DSM architecture.

Figure 1 illustrates a multi-host CXL-DSM architecture comprising multiple compute nodes (hosts) connected to a CXL memory node. Each host or memory node integrates a CXL/PCIe Root Complex (RC) that issues and receives messages over CXL links. The memory node contains a CXL/PCIe RC, a CXL controller, and one or more memory controllers connected to multiple memory devices [30, 50, 51]. The CXL controller manages connections and access to the attached memory. By allowing multiple hosts to attach concurrently, CXL-DSM enables cache-coherent data sharing and collaborative computation across hosts. Optional CXL switches [25, 30, 30] can be inserted between hosts and devices to realize even larger multi-host systems.

2.2 CXL-DSM Cache Coherence over CXL.mem

CXL-DSM supports multi-host cache coherence [27, 52] using a hierarchical, directory-based MESI protocol. Figure 2 illustrates a simplified organization of the CXL coherence architecture comprising two cooperating components: (i) a per-processor local coherence directory and (ii) a device coherence directory on the CXL memory node. The per processor directory records the local coherence state and the core IDs for each cacheline resident in that processor's cache (including both local memory and CXL memory). The device coherence directory records the coherence state and the processor IDs for each CXL memory cacheline that reside in processors' caches. Throughout this paper, without loss of generality, we assume that each host contains only one processor to simplify the description.

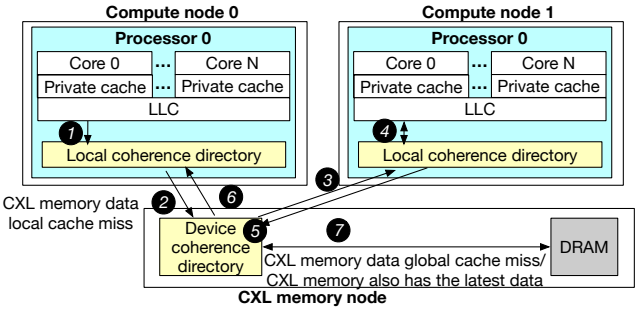

Figure 2. Coherence design of CXL-DSM.

For a CXL memory access, the local coherence directory first determines whether the requester processor's cache holds the most recent version of the target cache line. If not, this is considered a *local cache miss* and the request is forwarded to the device coherence directory on the CXL memory node. The device directory then checks whether any processor's cache holds the most recent copy; if none does, it is considered a *global cache miss* and the request is served from CXL memory. If some processors' cache holds shared-state clean copies, the request is still served from CXL memory, as accessing CXL memory incurs lower latency.

The workflow proceeds as follows. On a CXL memory access, the local coherence directory determines whether the requester processor's cache holds the most recent version of the target cache line ❶. If not, a *local cache miss* is declared and the request is forwarded to the device-side coherence

directory on the CXL memory node❷. If the device directory indicates that only another processor's cache holds the most recent copy, the memory node forwards the request to that processor❸; the processor, as validated by its local directory, supplies the cache line❹. The data is then returned to the requester, and both the device-side directory and the requester's local directory are updated❺❻. (3) If the device directory indicates that no processor cache holds a more recent copy, or if both CXL memory and a processor's cache hold the most recent copy, the request is serviced from CXL memory, and both directories are updated accordingly❼.

3 Motivation

3.1 Detailed Analysis of Multi-Host Migration

The CXL 3.1 standard and beyond [27] introduces the concept of *Global Integrated Memory* (GIM) [27, 53], allowing each host to expose part of its local memory into a global, unified memory address space. A host's page table can map a page that resides in its own local memory, in another host's local memory, or in CXL memory, thus allowing page migration between local memory and CXL memory. Inter-host accesses to another host's local memory [53, 54] are *non-cacheable to the requester host* [27, 53–55], thus always need to be routed through CXL root complexes, CXL links, and optional CXL switches. We present a simplified design consistent with CXL 3.1 to illustrate the page migration and access workflows.

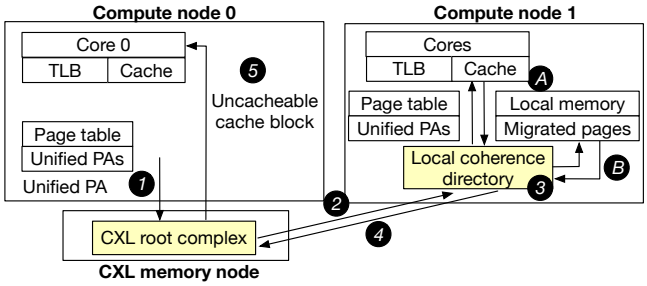

Figure 3. Workflow of accessing of migrated pages.

Workflow of inter-host access of migrated pages. Figure 3 ❶-❺ illustrates how a host accesses a page that has been migrated to another host's local memory. The local host processor first obtains the unified physical address (PA) from the TLB and page table and forwards it to the CXL Root Complex at the CXL memory node ❶. The root complex routes the request to the owning host indicated by the unified PA ❷. At the owning host, the local coherence directory is used to determine whether the most recent value resides in cache or in memory ❸; the data is then fetched into the owning node's LLC and returned to the CXL memory node ❹. The returned block, which is treated as non-cacheable at the requester host, is then delivered to the requester core. Serving this read miss requires a 4-hop traversal for the non-cacheable access. However, when the data resides in CXL memory, accesses are cacheable, which requires only two hops.

Take-away #1: In a multi-host CXL-DSM system, inter-host accesses to a migrated page are non-cacheable and require four hops. By contrast, accesses to CXL memory are cacheable and require at most two hops.

Workflow of local access of migrated pages. The non-cacheable access design increases the complexity of inter-host accesses but simplifies local accesses by eliminating coherence checks at the CXL memory node. As shown in Figure 3 Ⓐ Ⓑ, when a LLC read miss occurs, the unified PA is used to consult the local coherence directory to determine whether any cache within the host holds the most recent valid copy. If not, the request is served from local memory. As all inter-host accesses are non-cacheable, the design omits coherence probes to caches on other hosts, streamlining local-memory access.

Workflow of page migration. Page migration modifies a page's unified PA, which necessitates page table updates and TLB invalidations across all hosts. Each host uses its reserved page table to locate the process page tables that use the previous unified PA of the migrating page, and updates those entries to the new unified PA. Compared to single-host CXL disaggregated memory systems, this operation incurs a higher overhead in multi-host CXL-DSM because it requires broadcasting CXL RPCs [28] and performing more page table updates and TLB invalidations.

3.2 Quantitative Evaluation of Multi-Host Migration

Although recent research has investigated page migration policies and overhead optimization for single-host CXL disaggregated memory systems [11, 12, 39, 56–61], these approaches are ineffective in multi-host CXL-DSM due to their lack of awareness regarding the side effects of page migration and the poor migration scalability from higher demands in multi-host environments.

We evaluate existing page migration policies, originally designed for single-host CXL disaggregated memory systems, in a multi-host CXL-DSM environment to quantitatively assess the performance impact of the two limitations. Existing page migration policies can be broadly classified into recency-based [11, 39, 56, 62, 63] and frequency-based methods [8, 12, 64]. Specifically, we evaluate two state-of-the-art (SOTA) policies, Nomad (a recency-based method) [39] and Memtis (a frequency-based method) [12], using the Champinonship simulator [42, 43] configured as a four-host CXL-DSM system, with each host containing a single-socket CPU. Our evaluation employs memory-intensive benchmarks from prior studies, drawn from the GAP [65], PARSEC 3.0 [66], XSBench [67], YCSB [68, 69] and TPC-C [69] benchmark suites. Detailed evaluation settings are described in Section 5.1.

3.2.1 Side Effects of Page Migration on multi-host CXL-DSM.

We evaluated the percentage of harmful page

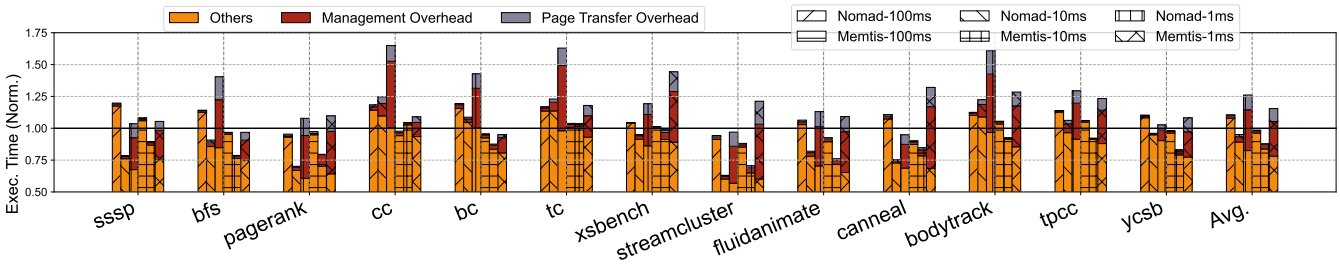

Figure 4. Performance breakdown with different page migration intervals, normalized to the no migration baseline.

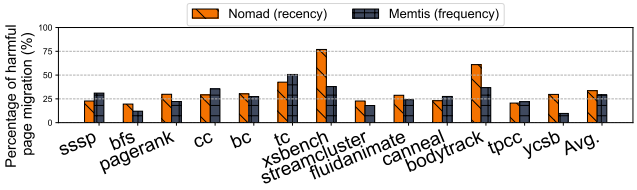

Figure 5. Percentage of harmful page migration.

migrations to understand the impact of neglecting the side effects of page migration in multi-host CXL-DSM. After a page is migrated from CXL memory to a host's local memory, subsequent accesses from that host transition from remote CXL memory access to local memory access. However, other hosts experience increased latency and non-cacheable accesses when referencing the migrated page compared to scenarios without migration. Thus, we define a page migration as harmful if it increases the overall execution time. We report the percentage of harmful page migrations observed in existing studies.

Figure 5 illustrates the percentage of harmful page migrations. On average, Nomad and Memtis exhibit 34% and 29% harmful migrations, respectively. These migrations negatively impact overall performance by increasing total execution time; refraining from performing such migrations would enhance performance. The increase in execution time arises because migrations convert accesses from other hosts into inter-host non-cacheable accesses, underscoring the importance of accounting for side effects in multi-host CXL-DSM page migration algorithms.

Take-away #2: Neglecting side effects in multi-host CXL-DSM page migration, existing migration techniques, such as Nomad and Memtis, result in 34% and 29% performance-degrading page migrations, respectively.

3.2.2 Poor Migration Scalability. Existing page migration techniques tailored for single-host systems generally adopt relatively long migration intervals (10 ms [8, 57] to a few seconds [11, 56, 62]) to balance migration overhead and performance benefits. However, the side-effects of page migration in multi-host CXL-DSM systems necessitate more efficient and timely migration mechanisms. We further conduct evaluations to quantitatively investigate: (1) whether multi-host CXL-DSM systems benefit from shorter page migration intervals (i.e., more timely and aggressive migration), and (2) the overhead breakdown associated with varying intervals. We report the performance breakdown, including page transfer overhead (data transfers incurred by migration), management overhead (e.g., page table updates and TLB invalidations), and other overheads.

Figure 4 presents the performance breakdown across three different page migration intervals (100 ms, 10 ms and 1ms), normalized against a no-migration baseline. The two state-of-the-art single-host migration methods show limited effectiveness in multi-host scenarios at the long interval (100 ms): Nomad increases execution time by 10.5% on average, while Memtis reduces it by only 1.4%. When adopting a shorter interval (10 ms) for more frequent page migration, execution time decreases by 4.8% and 12.2% on average. However, at the 1 ms interval, Nomad and Memtis increase execution time by 26.1% and 15.4% on average, respectively, due to the increased management overhead and page transfers.

Take-away #3: Multi-host CXL-DSM systems require shorter migration intervals to effectively capture page accesses from multiple hosts.

Take-away #4: At shorter intervals, page migration overhead becomes the dominant source of overhead, requiring efficient page migration.

3.3 Other Related Work

Several recent studies have investigated page migration in single-host CXL-disaggregated memory systems; however, their contributions does not address the previously discussed challenges associated with multi-host CXL-DSM page migration. They are orthogonal with the objectives pursued by our work. Specifically, Neomem [57] and M5 [58] offload hotness detection to the CXL memory side to facilitate efficient, low-latency access tracking. Colloid [40] balances memory placement between local and remote memory to minimize overall latency. Alto and Soar [38] employ MLP-aware policies to determine and dynamically adjust initial memory allocations across local and remote memory.

Intel Flat Mode [60, 70] is a recently introduced hardware-tiering technology designed for single-host CXL-disaggregated memory systems. Under this scheme, when a host accesses a cache block residing in CXL memory, the block is transparently swapped with a corresponding block in the host's local memory. However, Intel Flat Mode is incompatible with multi-host CXL-DSM. First, swapping memory lines between local memory and CXL memory switches the coherence domain between cache-coherent CXL-DSM and non-cacheable local memory, thereby violating coherence requirements. Second, Intel Flat Mode employs a static one-to-one mapping between CXL memory and local DRAM [60], which is impractical in multi-host environments where each host has distinct local DRAM regions. In our evaluation, we implement an Intel Flat Mode-like baseline (referred to as **HW-static**), utilizing parts of our design, to allow comparisons with hardware-tiering approaches.

4 Design

4.1 Overview

Based on the quantitative and qualitative analysis in Section 3, an effective and efficient page migration for multi-host CXL-DSM should consider the side effects of migrating data from CXL memory to local memory and reduce page migration overhead.

We attribute the inefficiency of existing single-host page migration methods to their **single-destination and rigid per-page migration**. Specifically, even when certain cache blocks within a page are frequently accessed by one host and other blocks are rarely accessed or predominantly accessed by other hosts, existing strategies either fully migrate the entire page or retain it entirely within CXL memory. This strategy fails to exploit optimization opportunities by treating different cache blocks within the same page separately (i.e., selectively migrating cache blocks). Additionally, per-page migration at low migration intervals incurs substantial overhead due to both management operations and data transfer costs.

We propose **Partial and Incremental Page Migration (PIPM)** for multi-host CXL-DSM. **Partial Migration** selectively migrates only frequently accessed cache blocks of a page to a host's local memory while leaving less-used blocks in CXL memory. This approach differentiates local and inter-host access patterns at fine granularity, mitigating side effects associated with per-page migration and significantly reducing management overhead (e.g., page-table updates and TLB invalidations). **Incremental Migration** leverages intrinsic memory accesses to incrementally migrate cache blocks upon cache eviction or writeback, avoiding explicit whole-page migrations and associated data-transfer overheads. Collectively, PIPM effectively addresses the previously identified challenges in multi-host memory management.

We develop architectural support to effectively and efficiently enable PIPM, facilitating transparent partial and incremental migration without requiring software modifications. As illustrated in Figure 6, our design introduces a per-host **Local Remapping Table** and a **Global Remapping Table** located on CXL memory to track pages undergoing partial migration. Specifically, the global remapping table records the migration destination host ID for each CXL-DSM page, while the local remapping table on each host stores the physical address mappings of CXL-DSM pages that migrate to the local memory of that host. We propose a **PIPM Majority-vote Migration Policy** that aggregates page-access information across multiple hosts, enabling globally optimized decisions regarding the necessity and placement of partially migrated pages. To ensure coherent access to partially migrated pages, we design the **PIPM Coherence** to incorporate partially migrated pages into the coherence domain, permitting incremental migration and cacheable access by other hosts.

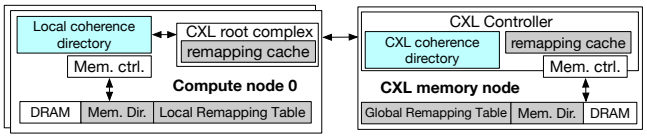

Figure 6. PIPM design overview.

4.2 PIPM Migration Policy

Existing page migration policies [5, 62, 71] are ineffective in multi-host CXL-DSM environments due to their neglect of migration side effects (**Takeaways #1 and #2**). To address this, PIPM introduces a hardware-based majority-vote migration policy inspired by the Boyer-Moore algorithm [72], enabling globally optimized decisions regarding the necessity and placement of partially migrated pages.

The intuition behind PIPM majority-vote migration policy is that partial migration is initiated only when the number of page accesses from a single host exceeds the combined accesses from all other hosts by a predefined threshold. This allows automatically making near-optimal data placement decisions regardless of the host count, retaining data in CXL-DSM when cross-host accesses are almost balanced, and migrating data into host DRAM when there is a clear vote winner. It is important to note that initiating partial migration solely involves updating the local and global remapping tables; thus, no page-table updates or TLB invalidations are required. The partial migration step only identifies the host to which cache blocks should be migrated, without triggering immediate data transfers.

Figure 7 illustrates the architectural components designed to support the PIPM migration policy. The global remapping cache records recently accessed CXL pages and is backed by an in-memory global remapping table. Each entry in the

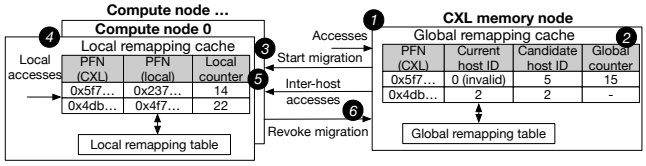

Figure 7. Partial migration workflow.

global remapping table records metadata for a CXL-DSM page, comprising a 5-bit current host ID, a 5-bit candidate host ID, and a 6-bit global counter. The local remapping table of each host only tracks pages partially migrated to that host. Each entry in the local remapping table contains a 28-bit PFN (indexing 1TB local DRAM) referring to as the page's PFN in local memory, and a 4-bit local counter.

The global and local counters implement the PIPM majority-vote migration policy for initiating and revoking partial migration, as described below: The global counter tracks whether a particular host (indicated by the candidate host ID) has more accesses than all other hosts to issue partial migration. Specifically, the global counter is incremented by one when the access originates from the candidate host and decremented by one when accessed by other hosts. When the global counter reaches zero, the next host to access the page updates the candidate host ID ❶. If the global counter reaches a predefined partial migration threshold ❷, partial migration of this page is initiated by creating an entry in the candidate host's local remapping table. The local PFN for this entry, allocated by the host's OS/hypervisor, identifies the location where partially migrated data from CXL memory is stored, and the entry's local counter is initialized to the migration threshold ❸. After a page has been partially migrated, its current host ID is set to that host's ID.

The local counter, stored in each host's local remapping table, records local accesses to partially migrated pages since local accesses bypass the global counter maintained at the CXL memory node ❹. Also, inter-host accesses decrement the local counter for that page ❺. If the local counter of a partially migrated page reaches zero, partial migration for the page is revoked by migrating all cache blocks from local memory back to their original CXL memory location, removing the corresponding entry from the local remapping table, and resetting the current host ID in the corresponding global remapping table entry ❻.

4.3 PIPM Coherence and Incremental Migration Design

In existing multi-host CXL-DSM systems, only the hosts' caches and CXL memory are within the coherence domain. The hosts' local memory lies outside this coherence domain, precluding our proposed PIPM approach, as partially migrated cache blocks in local memory cannot be accessed coherently and cacheably.

4.3.1 Naive Coherence Solution.

A straightforward solution is to introduce a 1-bit in-memory state for each cache block in the CXL memory to track partially migrated data[1]. This state indicates whether the associated cache block holds the most recent version; if it does not, the request will be redirected to the alternative memory (either local or CXL) to retrieve the latest data. However, this approach is inefficient for multi-host CXL-DSM because existing coherence protocols require completing a coherence state check for all caches in the CXL memory node and initiating a memory access from the CXL memory node—even if the latest version resides in local memory. This complexity arises from the potential for other hosts' caches to hold the latest version due to cacheable accesses.

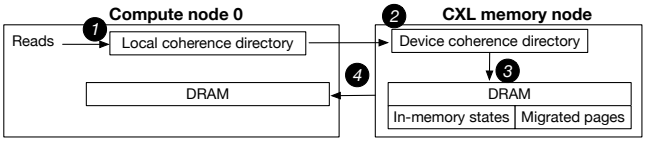

Figure 8. Read workflow of a naive coherence solution

Figure 8 illustrates the workflow of this naive coherence solution. A read access from the owning host to a partially migrated page first queries the local coherence directory to determine whether any caches within the host contain the most recent data ❶. If not found locally (i.e., Invalid state), the CXL device coherence directory is consulted to check whether the caches of other hosts hold the latest version ❷. If this also yields no result, the corresponding 1-bit in-memory state is examined. A value of 0 indicates that the most recent copy resides in CXL memory, performing data retrieval from there ❸; conversely, a value of 1 denotes that the latest copy is stored in local memory, performing retrieval from local storage ❹. Regardless of which host initiates the access or the location of the latest data for partially migrated pages, these steps must be executed, incurring substantial overhead that may negate the benefits of page migration for local accesses.

4.3.2 PIPM Coherence State Design.

The objective of PIPM coherence design is to ensure that all local accesses to a partially migrated page first query the local memory for the latest data before forwarding requests to the CXL memory node, and to enable incremental migration based on the most recent accessor (i.e., migrate to local DRAM if the most recent accessor is the local host, migrate back to

[1]For most server-grade DRAM, each memory line is augmented with additional ECC bits, which are fetched, verified, updated, or discarded together with the data upon every memory access. ECC typically occupies 8 bytes per line, providing several tens of spare bits. These bits have been leveraged as indices for memory remapping (e.g., Intel Flat Mode [60, 70]) or as in-memory states for maintaining NUMA cache coherence (e.g., Intel ccNUMA [73]).

CXL-DSM upon an inter-host access). To accomplish this, we redesign the coherence protocol and utilize 1-bit in-memory states in both local and CXL memory for partially migrated pages.

Extra States. Existing coherence protocols define M, S, and I states—representing Modified, Shared, and Invalid states—in both local coherence directories and device coherence directories. To realize our PIPM coherence design, we introduce an additional per-cache-block in-memory bit in both local and CXL memory, along with a new coherence state (**ME**) in the local coherence directory. By default, the in-memory bit is initialized to 0. When a cache block migrates to local DRAM, this bit is set to 1 in both the local DRAM and CXL-DSM. The coherence directory state combined with the in-memory bit collectively defines the PIPM coherence state of a cache block.

In the local coherence directory, the newly introduced **ME state** (Migrated-Modified/Exclusive) indicates that the corresponding cache block has been migrated to the local memory of the host and is cached exclusively in this host's cache. Subsequent local accesses to cache blocks in the ME state can proceed without querying the device coherence directory, thus enabling efficient coherence handling. The encoding for the ME state comprises a new ME state in the local coherence directory paired with an in-memory bit set to 1, as illustrated in the upper table of Figure 9. Additionally, we introduce the **I' state** (Migrated-Invalid), representing that the cache block is migrated to the local memory of the host but not cached (i.e., Invalid in the directory). The encoding for the I' state reuses the invalid (I) state in the local coherence directory combined with an in-memory bit set to 1, as depicted in the upper table of Figure 9.

In the device coherence directory, we also introduce the **I' state** to indicate that the corresponding cache block has been migrated to a host's local memory. Inter-host accesses to cache blocks marked as I' in the device coherence directory must be directed to the host's local memory. The encoding of the I' state reuses the Invalid (I) state in the device coherence directory in conjunction with an in-memory bit set to 1, as illustrated in the lower table of Figure 9.

4.3.3 PIPM Coherence State Transition.

The right side of Figure 9 illustrates the PIPM coherence state transitions triggered by various events, including six newly introduced transitions: local writeback operations (case ❶) that initiate incremental migration from CXL memory to a host's local memory; inter-host reads and writes (cases ❷, ❺, and ❻) that trigger incremental migration from local memory back to CXL memory; and efficient local memory accesses (cases ❸ and ❹). For clarity and simplicity, the standard coherence request handling workflow [73, 74], which remains unchanged, is omitted from the following description.

Case ❶: Incremental Migration upon Local WriteBack (`Loc-WB`). When the local directory state is M, it indicates that the local node was the most recent accessor of the cache block (otherwise, the state would be either S or I) and that the block has not yet been migrated into local memory (otherwise, it would be ME). Under these conditions, a writeback operation triggers incremental migration. This migration process involves invalidating the corresponding entries in both the host and CXL coherence directories as well as the host's cache entry, retrieving and flipping the associated in-memory state bits in both local and CXL memory, and subsequently performing the incremental migration. Upon completion, the coherence state transitions from M to I' in both the local host directory and the CXL device directory.

Case ❸ and ❹: Local Accesses (`Loc-Rd/Loc-Wr/Loc-WB`) to Migrated Cache Blocks. Once a cache block has been migrated to local memory, ❸ subsequent local memory requests are served directly from local memory, with the host coherence directory updated accordingly (transitioning from I' to ME). Consequently, the CXL directory no longer needs to allocate an entry for this cache block, thereby eliminating unnecessary host-device CXL traffic. ❹ When this cache block is subsequently evicted from the local cache (transitioning from ME back to I'), only a dirty data writeback and invalidation of the corresponding host directory entry are required.

Case ❷: Migration back to CXL-DSM upon inter-host memory accesses (`Inter-Rd/Inter-Wr`) in I' State. When no valid cache copies exist (i.e., the migrated cache block is in the I' state on both the host and device sides), another host's CXL memory access to the migrated line is directed to the CXL device directory. The CXL directory issues a CXL memory read to verify the I' coherence state, after which the request is forwarded to the local directory of the host currently owning the migrated data. The migrated host's local directory retrieves both the memory line and the associated in-memory bit, then performs an asynchronous memory writeback, updating its coherence state from I' to I. Upon receiving this response, the CXL directory allocates a directory entry for the cache line and updates its state to M. Finally, the retrieved data is cached in the requester host's cache.

Case ❺ and ❻: Migration back to CXL memory upon inter-host accesses (`Inter-Rd/Inter-Wr`) in ME state. When a migrated cache line is exclusively cached at the local host (i.e., in the ME state on the host side and the I' state on the device side), inter-host accesses are still routed through the requester host's directory, the CXL directory, and finally the owning host's local directory. The owning host's local directory subsequently updates its coherence state—transitioning from ME to I for ❺ `Inter-Wr`, or from ME to S for ❻ `Inter-Rd`—and initiates an asynchronous memory writeback to update the in-memory state bit. Upon receiving the response, the CXL directory allocates an entry and updates its coherence state accordingly: from I to M for case ❺, or from I to S for case ❻. Finally, the requested data is cached in the requester host's cache.

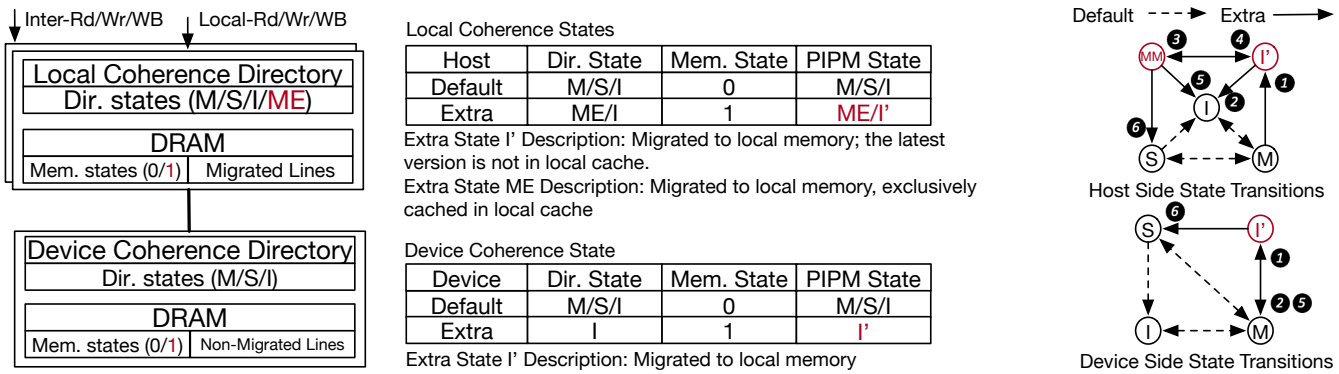

Figure 9. PIPM coherence design.

Interaction with global and local remapping tables. PIPM requires accessing global and local remapping tables only for shared data access. For local private data (i.e., data allocated (and pinned) in local DRAM for security or performance considerations) access, PIPM does not introduce any remapping table lookups or coherence request handling modifications. When initiating a memory request, existing processors that support CXL first perform a simple physical address range check to route the memory request to the local memory controller or the CXL RC accordingly. As accesses to shared data always carry physical addresses within the CXL-DSM physical address range *regardless of whether the shared data pages are partially migrated or not* after virtual-to-physical address translation and before remapping table lookup, processors can always distinguish local private data accesses from shared data accesses after the range check.

For shared data access, on each LLC miss (i.e., when the local coherence directory is in I state), the requester needs to first perform a local remapping table lookup to retrieve the full local coherence state (I or I'). Also, each migrated memory line access requires a local remapping table lookup. Global remapping table access occurs only when forwarding remote access requests (case ❷, ❺ and ❻).

4.4 Space Overhead

The local remapping table on each host's DRAM requires 4 Bytes per entry to store a 28-bit PFN (capable of indexing up to 1TB of local DRAM) and a 4-bit access counter. It is organized as a two-level radix page table [75, 76] with a fixed root node size of 32MB (8 Bytes per entry, indexing up to 4M page table pages, where each PT page stores 1K page table entries) to balance access latency and storage overhead. It requires only (32MB + 4B/4KB × RSS), which is approximately 0.1% of the total resident Set Size (RSS) of the workloads. The global remapping table in CXL-DSM requires only 2 Bytes per entry (consisting of a 5-bit current ID, a 5-bit candidate ID, and a 6-bit access counter), accounting for just 0.05% of the total CXL-DSM size. By default, PIPM requires only a 16KB global remapping cache on the CXL device and a 1MB local remapping cache on each host's RC to effectively cache remapping entries.

5 Evaluation

5.1 Evaluation Methodology

5.1.1 Benchmarks. Our target large-scale multi-host systems typically run memory-intensive workloads with large memory footprints that do not fit within a single socket and large working set sizes that significantly exceed on-die LLC capacities. Following prior work [71, 77–80], we select representative large-scale, memory-intensive workloads, as listed in Table 1.

Table 1. Evaluated workloads.

Benchmark	Benchmark Suite	Memory Footprint
SSSP (Single-Source Shortest Paths)	GAPBS [65] (Kron)	48GB
BFS (Breadth-first Search)	GAPBS	48GB
PR (Compute the PageRank score)	GAPBS	48GB
CC (Connected components)	GAPBS	48GB
BC (Betweenness centrality)	GAPBS	48GB
TC (Triangle Counting)	GAPBS	48GB
XSBench (Computational kernel of the Monte Carlo neutron transport algorithm)	XSBench [67]	42GB
streamcluster (Data stream clustering)	PARSEC [66]	18GB
fluidanimate (Fluid simulation)	PARSEC	10GB
canneal (Annealing simulation)	PARSEC	12GB
bodytrack (Annealed particle filter)	PARSEC	8GB
TPC-C (Default) (Transaction)	Silo [69]	24GB
YCSB (R:W 4:1) (Database)	Silo	15GB

5.1.2 Simulation Methodology. We model the multi-host CXL-DSM architecture using a cycle-level, trace-based timing simulator [42, 43]. The simulator configuration is detailed in Table 2. Following prior works [71, 81], our simulation methodology consists of the following steps: (1) We first execute the target workloads on real hardware and use Intel Pintool [82] to collect instruction and memory traces. (2) We

then replay the collected memory traces on the simulator to generate memory mapping checkpoints at every 1-billion-instruction interval. (3) Finally, we perform detailed core simulation, beginning after a warm-up phase, utilizing the corresponding checkpoints and traces. This methodology enables the simulation of applications with memory footprints on the order of tens of gigabytes and sufficiently long runtimes (10 billion instructions per core).

5.1.3 Compared Schemes. We compare PIPM against the following related works: (1) **Native CXL-DSM**, the baseline configuration that does not support data migration to hosts' local memory; (2) **Nomad**[39], which employs a state-of-the-art recency-based hotness migration policy[11, 56] and optimizes kernel-based page migration by enabling asynchronous migration; (3) **Memtis**[12], utilizing a state-of-the-art frequency-based hotness migration policy; and (4) **HeMem**[8], another frequency-based hotness migration method. We also introduce two ablation baselines to separately analyze the effectiveness of PIPM's migration policy and mechanism: (5) **OS-skew**, which combines the PIPM migration policy with a conventional kernel-based migration mechanism; and (6) **HW-static**, which employs incremental migration enabled by the PIPM coherence protocol but with a static mapping strategy (i.e., without our adaptive migration policy), analogous to prior hardware-tiering approaches such as Intel Flat Mode [60, 70]. Under HW-static, CXL-DSM is uniformly partitioned and statically mapped to each host's local memory. We also include an upper-bound estimation, (7) **Ideal**, where the workloads run on a single-socket CPU with sufficiently large DRAM to hold all data.

Table 2. Scaled-down System Configuration.

Architecture	4 hosts, 1 single-socket CPU each host
CPU	4 OoO cores, 4GHz, 6-wide, 224-entry ROB, 72-entry LQ, 56-entry SQ
Private L1-(I/D)	32KB, 8-way, 4 cycle RT (round-trip) latency
Shared LLC	2MB per core, 16-way, 24-cycle RT latency
DRAM	2x DDR5-4800 channels 128GB CXL-DSM; 1x DDR5-4800 channel 32GB DRAM per host
tRC-tRCD-tCL-tRP	48-15-20-15
CXL link	latency: 50ns, bandwidth: 5GB/s (per direction)
CXL Directory	2048-set, 16-way per slice, 4 slices, 32-cycle RT latency, 2GHz
PIPM parameters	16KB 8-way global remapping cache, 4-cycle RT; 1MB 8-way local remapping cache, 8-cycle RT; migration threshold: 8

5.1.4 Correctness and Implementation. We implement the PIPM cache coherence protocol on top of the MSI protocol and verify it using the model checking tool Murφ [83], proving that PIPM coherence does not incur any deadlock, and does not violate conceived Single-Writer-Multiple-Reader (SWMR) invariant and Sequential Consistency (SC) model. For simulation, we implement packet-level coherence behaviors for both default CXL-DSM and the PIPM coherence protocol using a locked-based scheme similar to ZSim [84]'s implementation. Based on this, we are able to model full system cache coherency including per-core private cache, and both on-chip and off-chip network traffic. For all evaluation, we assume the code segment, kernel components (e.g., page tables), and thread stacks are treated as private local data, while heap data (e.g., database instances, graphs) are shared across hosts. Following prior work about multi-host CXL-DSM [2, 30, 33], we initially place all shared data in CXL-DSM.

For page migration schemes, we assume a 20μs 4KB migration-induced overhead for the initiating core [5, 85], a 5μs overhead for other cores, a 10ms migration interval [8] and apply optimizations such as batching TLB shootdowns [86, 87] and multi-threaded, batched page transfers [5] to reduce page migration overhead. For PIPM, migration decisions are made immediately upon exceeding the promotion threshold, as it incurs no kernel-induced overhead or whole-page transfers. We empirically set migration thresholds for both PIPM (where we observe similar performance with threshold ranging from 4 to 16) and baseline schemes for the best performance.

5.2 End-to-end Performance

5.2.1 Overall Performance. Figure 10 presents the overall performance of all evaluated schemes normalized to the *Native CXL-DSM* baseline. PIPM outperforms the other schemes across all workloads, achieving an average performance of 1.86× and 0.73× (up to 2.65× and 0.94×) compared to *Native CXL-DSM* and *Ideal*, respectively, underscoring its substantial performance benefits. Specifically, graph analytics workloads such as SSSP and PageRank, where worker threads independently access memory with strong locality patterns (e.g., adjacency matrix traversals), demonstrate significant performance improvements ranging from 142% to 151%. Database workloads such as TPC-C and YCSB, characterized by random and scattered user-thread accesses, yield more modest performance gains (36%–53%). In contrast, existing page migration schemes employing traditional hotness-based policies (*Nomad*, *Memtis*, and *HeMem*) achieve only marginal improvements over *Native CXL-DSM* and even degrade performance by up to 18% in five workloads. This inefficiency arises because these single-host-oriented designs neither account for migration-induced side effects nor optimize migration overhead, significantly restricting performance potential of page migration in multi-host CXL-DSM scenarios.

5.2.2 Ablation. The *OS-skew* baseline, despite employing the PIPM migration policy, achieves only a 31.5% average improvement over *Native CXL-DSM* due to its inefficient and rigid page-migration mechanism. The *HW-static* baseline

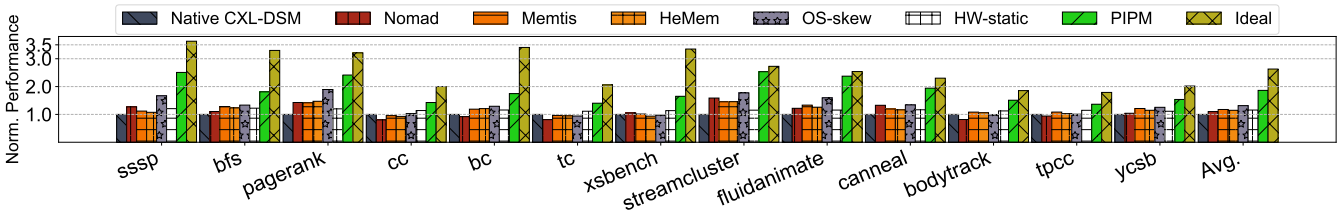

Figure 10. End-to-end performance normalized to Native CXL-DSM.

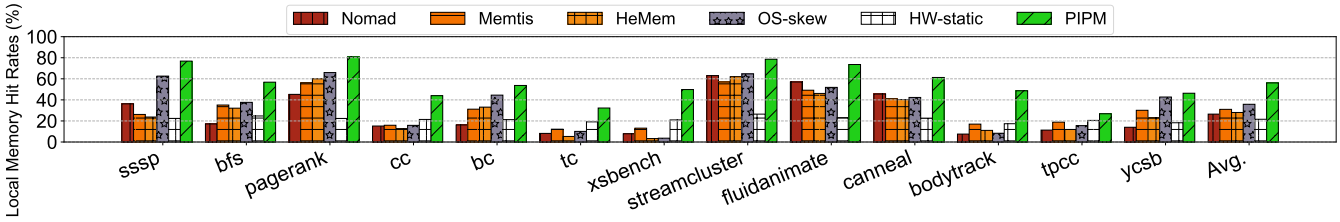

Figure 11. Local memory hit rates.

leverages hardware-based incremental cache block migration via the PIPM coherence protocol but employs a fixed, static mapping between CXL-DSM and each host's local memory. Consequently, data blocks benefiting from local caching may be inefficiently mapped into other hosts' memory, substantially limiting potential performance gains from fine-grained migration. As a result, *HW-static* yields a modest average improvement of only 15.7% over *Native CXL-DSM*. Overall, PIPM surpasses both *OS-skew* and *HW-static* by an average of 41.7% and 61.1%, respectively. These results demonstrate that both the partial incremental migration mechanisms and the PIPM migration policy are critical for achieving effective memory management for multi-host CXL-DSM systems.

5.3 Performance Analysis

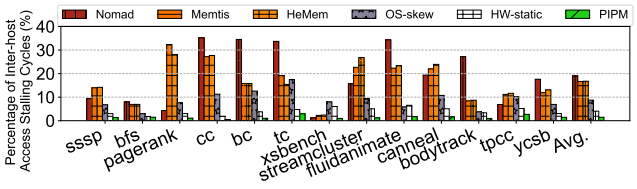

Figure 12. Stalling cycles of inter-host memory access normalized to native CXL-DSM total execution time.

5.3.1 Memory Access Characteristics. To further investigate the effectiveness of PIPM's migration mechanism, we evaluate both the local memory hit ratio and the contribution of inter-host memory access stalls to the total execution time for all schemes.

Figure 11 presents the local memory hit rates across all schemes, where misses are directed to either CXL memory or another host's memory. PIPM achieves a local memory hit rate of 56.1% on average, significantly outperforming *Nomad* (26.5%), *Memtis* (31.0%), and *HeMem* (28.1%). *OS-skew* exhibits a relatively higher local hit rate due to its use of the PIPM migration policy.

Figure 12 illustrates the contribution of stalling cycles from inter-host memory accesses to overall execution time. *Nomad*, *Memtis*, and *HeMem* incur higher stall contributions (averaging 19.1%, 16.6%, and 16.8%, respectively) due to their whole-page migration strategies, which hinder rapid data migration between host memory and CXL memory, thus increasing inter-host memory access frequency. *OS-skew* achieves lower stall contributions from inter-host memory accesses (8.7% on average) owing to the PIPM migration policy, which effectively prevents migration of pages into a host's memory when there are frequent accesses from other hosts.

HW-static induces fewer inter-host memory accesses than kernel-based baselines, contributing only 4.1% to total execution time. However, as shown in Figure 11, it also results in a lower local memory access ratio (21.6% on average), due to its inability to dynamically remap data to hosts that could better utilize local memory. In contrast, PIPM demonstrates the lowest inter-host memory access stall overhead (only 1.5% of total execution time) while simultaneously maintaining the highest local memory access ratio (56.1% on average).

5.3.2 Memory Consumption. Figure 13 illustrates the average ratios of local memory footprint per host to the total memory footprint. Traditional hotness-based migration policies (*Nomad*, *HeMem*, and *Memtis*) migrate frequently accessed pages into local memoruy without considering inter-host memory access, resulting in average per-host memory allocations of 7.4%, 6.0%, and 5.2%, respectively. In contrast,

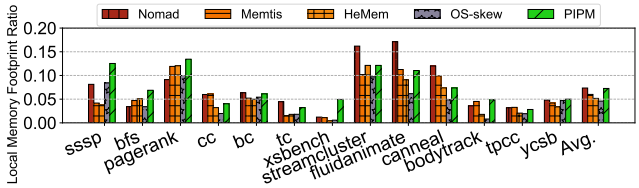

Figure 13. Average ratios of local memory footprint per host to total memory footprint.

OS-skew selectively migrates pages to local memory, thereby reducing its average per-host allocation to 4.6%. The *HW-static* baseline employs a static 1:1 mapping strategy (similar to Intel Flat Mode [60, 70]), lacking dynamic remapping capability and thus maintaining a fixed local memory allocation of 25% per host. In comparison, PIPM leverages both its migration policy and partial incremental migration mechanism, achieving a modest average allocation of only 7.3% per host, while significantly enhancing local memory hit rates and overall system performance.

5.4 Sensitivity Study and Scalability

5.4.1 Sensitivity to CXL Link Latency. Figure 14 shows the relative performance improvement of PIPM over *Native CXL-DSM* under different CXL link latencies. At a higher link latency of 100ns per direction (representative of configurations with a CXL switch), PIPM achieves an additional performance improvement of 55.7% on average (up to 193.1%), as the benefits of local memory access become more pronounced.

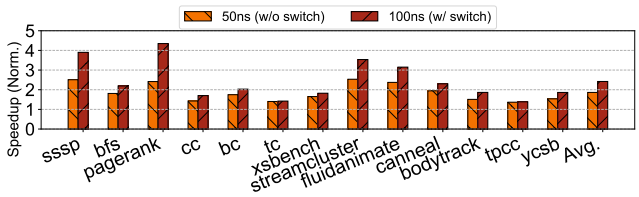

Figure 14. Overall IPC Performance Speedup over Native CXL-DSM under Different CXL Link Latencies.

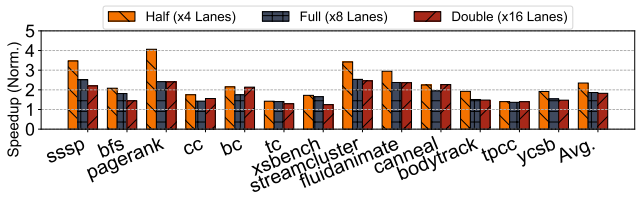

Figure 15. Overall IPC Performance Speedup over Native CXL-DSM under Different CXL Link Bandwidths.

5.4.2 Sensitivity to CXL Link Bandwidth. We use an 8x scaled-down setting as the default configuration (32 cores ⇒ 4 cores per host, 64 GB/s (40 GB/s effective [71]) ⇒ 8 GB/s (5 GB/s) over x16 CXL lanes). As shown in Figure 15, with half the bandwidth (x8 CXL lanes), PIPM achieves an 48.4% (up to 96%) performance gain over Native CXL-DSM relative to the x16 lanes setting, as most applications become both bandwidth- and latency-bound and thus benefit more from partial incremental migration. With 2x bandwidth (x32 CXL lanes), PIPM retains 97.9% of the relative performance improvement over *Native CXL-DSM* achieved under the x16 lanes setting, demonstrating that most workloads still significantly benefit from partial incremental migration due to their latency-bound characteristics.

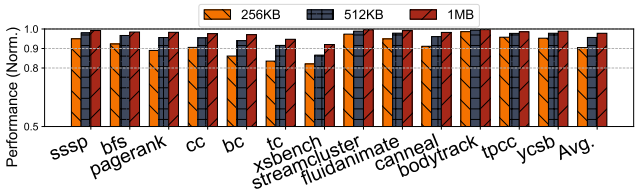

Figure 16. Performance of different local remapping cache Sizes, normalized to infinite local remapping cache size.

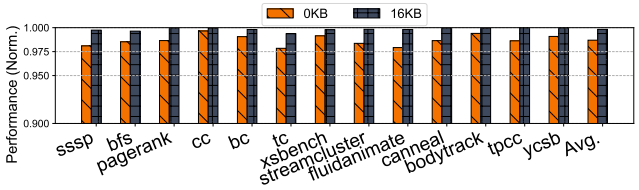

Figure 17. Performance of different global remapping cache sizes, normalized to infinite global remapping cache size.

5.4.3 Sensitivity to Area Overhead. We vary the on-die buffer capacities of both the local remapping cache and the global remapping cache to evaluate their impact on end-to-end performance. As shown in Figure 16 and Figure 17, the local remapping cache capacity has a higher impact on overall performance, as local remapping table lookups are on the critical path of local memory accesses, whereas global remapping table accesses occur only on inter-host memory accesses. We observe that a 16KB global remapping cache is sufficient to achieve 99.8% of the performance of an ideal infinite global remapping cache, while a 1MB local remapping cache per host achieves 97.8% of the performance of an ideal infinite local remapping cache. *Overall, the area overhead of PIPM is negligible*, requiring only a 1MB local remapping cache per host on the RC, and a 16KB global remapping cache on the CXL device.

6 Related Work

In addition to the related work discussed in Section 2 and Section 3, this section covers other related studies.

Application-level Optimization over CXL-DSM. Recent works [29, 30, 32, 34, 88–90] focus on application-level optimizations for (CXL-DSM-based) large shared memory pools, including SW prefetching [34, 88], SW-managed coherence [29, 30, 89], replications [29, 89]. *PIPM is orthogonal to these works and can even further support application-level optimizations by exposing interfaces to software.* For example, applications can leverage PIPM's line-level migration to enable fine-grained, lock-free prefetching, or explicitly enable or disable incremental migration for specific pages based on program semantics to improve performance. Also, the PIPM coherence can potentially mitigate the on-die area overhead of the CXL coherence directory [32, 91] for supporting CXL 3.0 multi-host coherence, as migrated memory lines no longer require allocating CXL directory entries until they are migrated back to CXL-DSM.

Automatic Memory Management. A large number of prior works explore page management for tiered memory systems [8, 11, 12, 39, 57–61, 64, 70, 92–95] and NUMA systems [56, 62]. In contrast, PIPM targets multi-host CXL-DSM systems. PIPM tackles the inefficiency of existing page migration schemes over multi-host CXL-DSM systems by enabling meticulously combining a coherence-aware, incremental migration mechanism with page-level migration policy, PIPM tackles the inefficiency of existing page migration schemes over multi-host CXL-DSM systems while maintaining low overhead.

Distributed Shared Memory Systems. Previous distributed shared memory systems [96–99] rely on interconnects with socket-like interfaces (e.g., RDMA). They typically employ page-based block granularity and locked-based software cache coherency with manually managed data placement. With the emerging CXL interconnects and hardware cache-coherent CXL-DSM introduced in CXL 3.x, distributed shared memory systems are able to support more efficient, finer-grained data management at rack scale with less software modification. Our work built on top of CXL-DSM proposes architectural support to further unlock the potential of CXL for distributed shared memory systems.

7 Conclusion

We propose **Partial and Incremental Page Migration (PIPM)** for multi-host CXL-DSM, which selectively migrates frequently accessed cache blocks into local memory and incrementally transfers data using intrinsic memory accesses. We develop architectural support including global and local remapping tables, PIPM migration policy, and PIPM coherence protocol to effectively enable partial and incremental page migration. Evaluations show PIPM achieves up to 2.54× (1.86× average) speedup over existing methods, systematically overcoming key limitations of multi-host CXL-DSM.

Acknowledgment

We would like to thank the anonymous reviewers from ASPLOS 2026 for their insightful and constructive feedback.

References

[1] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *ACM transactions on intelligent systems and technology*, 15(3):1–45, 2024.

[2] Yibo Huang, Haowei Chen, Newton Ni, Vijay Chidambaram, Dixin Tang, Emmett Witchel, Zhiting Zhu, and Zhipeng Jia. Tigon: A distributed database for a cxl pod. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25), Boston, MA*, 2025.

[3] Chun-Wei Tsai, Chin-Feng Lai, Han-Chieh Chao, and Athanasios V Vasilakos. Big data analytics: a survey. *Journal of Big data*, 2(1):21, 2015.

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

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

[6] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. {LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, 2018.

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

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

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

[10] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, et al. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 727–741, 2023.

[11] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 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*, pages 742–755, 2023.

[12] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium*

on Operating Systems Principles, pages 17–34, 2023.

[13] Baptiste Lepers and Willy Zwaenepoel. Johnny cache: the end of {DRAM} cache conflicts (in tiered main memory systems). In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 519–534, 2023.

[14] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332, 2020.

[15] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A {Memory-Disaggregated} managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280, 2020.

[16] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. Klocs: Kernel-level object contexts for heterogeneous memory systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 65–78, 2021.

[17] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 417–433, 2022.

[18] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H Peter Hofstee. 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)*, pages 868–880. IEEE, 2020.

[19] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

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

[21] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 317–330, 2019.

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

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

[24] Onur Mutlu. Memory scaling: A systems architecture perspective. In *2013 5th IEEE International Memory Workshop*, pages 21–25. IEEE, 2013.

[25] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 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*, pages 574–587, 2023.

[26] nextplatform.com. Cxl and gen-z iron out a coherent interconnect strategy. https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/, 2024. Online; accessed Jun, 2024.

[27] CXL. Cxl 3.1 specification. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.1-Specification.pdf, 2024. Online; accessed Jun, 2024.

[28] Teng Ma, Zheng Liu, Chengkun Wei, Jialiang Huang, Youwei Zhuo, Haoyu Li, Ning Zhang, Yijin Guan, Dimin Niu, Mingxing Zhang, et al. {HydraRPC}:{RPC} in the {CXL} era. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 387–395, 2024.

[29] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. Polardb-mp: a multi-primary cloud-native database via disaggregated shared memory. In *Companion of the 2024 International Conference on Management of Data*, pages 295–308, 2024.

[30] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Gerry Fan, Yang Kong, Bo Wang, Jing Fang, Yuhui Wang, Tao Huang, Wenpu Hu, Jim Kao, and Jianping Jiang. Unlocking the potential of cxl for disaggregated memory in cloud-native databases. In *Companion of the 2025 International Conference on Management of Data*, SIGMOD/PODS '25, page 689–702, New York, NY, USA, 2025. Association for Computing Machinery.

[31] Hooyoung Ahn, Seonyoung Kim, Yoomi Park, Woojong Han, Shinyoung Ahn, Tu Tran, Bharath Ramesh, Hari Subramoni, and Dhabaleswar K Panda. Mpi allgather utilizing cxl shared memory pool in multi-node computing systems. In *2024 IEEE International Conference on Big Data (BigData)*, pages 332–337. IEEE, 2024.

[32] Zhao Wang, Yiqi Chen, Cong Li, Dimin Niu, Tianchan Guan, Zhaoyang Du, Xingda Wei, and Guangyu Sun. Enabling efficient transaction processing on CXL-based memory sharing, 2025.

[33] Chloe Alverti, Stratos Psomadakis, Burak Ocalan, Shashwat Jaiswal, Tianyin Xu, and Josep Torrellas. Cxlfork: Fast remote fork over cxl fabrics. pages 210–226, 2025.

[34] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. CXL-ANNS: Software-Hardware collaborative memory disaggregation and computation for Billion-Scale approximate nearest neighbor search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 585–600, Boston, MA, July 2023. USENIX Association.

[35] Tu Tran, Mustafa Abduljabbar, Hooyoung Ahn, Seonyoung Kim, Yoomi Park, Woojong Han, Shinyoung Ahn, Hari Subramoni, and Dhabaleswar K. Panda. Omb-cxl: A micro-benchmark suite for evaluating mpi communication utilizing compute express link memory devices. In *Practice and Experience in Advanced Research Computing 2024: Human Powered Computing*, PEARC '24, New York, NY, USA, 2024. Association for Computing Machinery.

[36] Steve Abraham. Amazon aurora multi-master: Scaling out database write performance. 2016.

[37] Alex Depoutovitch, Chong Chen, Per-Ake Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, Yuchen Zhang, and Calvin Sun. Taurus mm: Bringing multi-master to the cloud. *Proc. VLDB Endow.*, 16(12):3488–3500, August 2023.

[38] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. Tiered memory management beyond hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 731–747, 2025.

[39] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad:{Non-Exclusive} memory tiering via transactional page migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 19–35, 2024.

[40] Midhul Vuppalapati and Rachit Agarwal. Tiered memory management: Access latency is the key! In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 79–94, New York, NY, USA, 2024. Association for Computing Machinery.

[41] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S Berger, Marie Nguyen, Xun Jian, Sam H Noh, and Huaicheng Li. Systematic cxl memory characterization and performance analysis at scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1203–1217, 2025.

[42] Nathan Gober, Gino Chacon, Lei Wang, Paul V Gratz, Daniel A Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. The championship simulator: Architectural simulation for education and competition. *arXiv preprint arXiv:2210.14324*, 2022.

[43] Champsim. https://github.com/ChampSim/ChampSim, 2025.

[44] nextplatform.com. A coherent interconnect strategy: Cxl absorbs gen-z. https://www.nextplatform.com/2021/11/23/finally-a-coherent-interconnect-strategy-cxl-absorbs-gen-z/, 2024. Online; accessed Jun, 2024.

[45] nextplatform.com. Pci-express 5.0: The unintended but formidable datacenter interconnect. https://www.nextplatform.com/2021/02/03/pci-express-5-0-the-unintended-but-formidable-datacenter-interconnect//, 2024. Online; accessed Jun, 2024.

[46] blocksandfiles.com. Intel sees cxl as rack-level disaggregator with optane connectivity. https://blocksandfiles.com/2021/08/18/intel-sees-cxl-as-rack-level-disaggregator/, 2024. Online; accessed Jun, 2024.

[47] Samsung. Samsung cxl solutions – cmm-h. https://semiconductor.samsung.com/us/news-events/tech-blog/samsung-cxl-solutions-cmm-h/, 2024. Online; accessed Jun, 2024.

[48] Samsung. Samsung unveils industry-first memory module incorporating new cxl interconnect standard. https://news.samsung.com/global/samsung-unveils-industry-first-memory-module-incorporating-new-cxl-interconnect-standard, 2024. Online; accessed Jun, 2024.

[49] SK Hynix. Sk hynix develops ddr5 dram cxltm memory to expand the cxl memory ecosystem. https://news.skhynix.com/sk-hynix-develops-ddr5-dram-cxltm-memory-to-expand-the-cxl-memory-ecosystem/, 2024. Online; accessed Jun, 2024.

[50] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, et al. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–121, 2023.

[51] CXL. Compute express link. https://computeexpresslink.org/, 2024. Online; accessed Jun, 2024.

[52] Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. Memory sharing with cxl: Hardware and software design approaches. *arXiv preprint arXiv:2404.03245*, 2024.

[53] Marks Kevin. Cxl for storage. https://www.snia.org/sites/default/files/SDC/Austin/SNIA-RSDC24-Marks-CXL-for-Storage.pdf, 2024.

[54] Yanpeng Yu, Nicolai Oswald, and Anurag Khandelwal. CORD: Low-latency, bandwidth-efficient and scalable release consistency via directory ordering. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 1311–1326. ACM, 2025.

[55] Houxiang Ji, Srikar Vanavasam, Yang Zhou, Qirong Xia, Jinghan Huang, Yifan Yuan, Ren Wang, Pekon Gupta, Bhushan Chitlur, Ipoom Jeong, and Nam Sung Kim. Demystifying a cxl type-2 device: A heterogeneous cooperative computing perspective. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1504–1517, 2024.

[56] Ying Huang and Hasan Al Maruf. https://lwn.net/Articles/876993/, 2021. [PATCH 0/5] Transparent Page Placement for Tiered-Memory.

[57] Zhe Zhou, Yiqi Chen, Tao Zhang, Yang Wang, Ran Shu, Shuotao Xu, Peng Cheng, Lei Qu, Yongqiang Xiong, Jie Zhang, et al. Neomem: Hardware/software co-design for cxl-native memory tiering. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1518–1531. IEEE, 2024.

[58] Yan Sun, Jongyul Kim, Zeduo Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. M5: Mastering page migration and memory management for cxl-based tiered memory systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 604–621, New York, NY, USA, 2025. Association for Computing Machinery.

[59] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. {FlexMem}: Adaptive page profiling and migration for tiered memory. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 817–833, 2024.

[60] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with cxl in virtualized environments. In *Symposium on Operating Systems Design and Implementation*, 2024.

[61] Xinyue Yi, Hongchao Du, Yu Wang, Jie Zhang, Qiao Li, and Chun Jason Xue. ArtMem: Adaptive Migration in Reinforcement Learning-Enabled Tiered Memory. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 405–418, Tokyo Japan, June 2025. ACM.

[62] Ying Huang. autonuma: Optimize page placement for memory tiering system - patchwork. 2024.

[63] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. Multi-clock: Dynamic tiering for hybrid memory systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 925–937, 2022.

[64] Kevin Song, Jiacheng Yang, Sihang Liu, and Gennady Pekhimenko. Lightweight frequency-based tiering for cxl memory systems. *arXiv preprint arXiv:2312.04789*, 2023.

[65] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[66] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. Parsec3. 0: A multicore benchmark suite with network stacks and splash-2x. *ACM SIGARCH Computer Architecture News*, 44(5):1–16, 2017.

[67] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[68] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[69] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[70] Michael D. Powell, Patrick Fleming, Venkidesh Iyer Krishna, Naveen Lakkakula, Subhiksha Ravisundar, Praveen Mosur, Arijit Biswas, Pradeep Dubey, Kapil Sood, Andrew Cunningham, and Smita Kumar. Intel xeon 6 product family. *IEEE Micro*, 45(3):31–40, 2025.

[71] Albert Cho and Alexandros Daglis. StarNUMA: Mitigating NUMA Challenges with Memory Pooling . In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 997–1012, Los Alamitos, CA, USA, Nov 2024. IEEE Computer Society.

[72] Robert S Boyer and J Strother Moore. Mjrty—a fast majority vote algorithm. In *Automated reasoning: essays in honor of Woody Bledsoe*, pages 105–117. Springer, 1991.

[73] Kevin Loughlin, Stefan Saroiu, Alec Wolman, Yatin A Manerkar, and Baris Kasikci. Moesi-prime: preventing coherence-induced hammering in commodity workloads. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 670–684, 2022.

[74] Daniel Sorin, Mark Hill, and David Wood. *A primer on memory consistency and cache coherence*. Springer Nature, 2022.

[75] Cristan Szmajda and Gernot Heiser. Variable radix page table: A page table for modern architectures. In *Asia-Pacific conference on advances in computer systems architecture*, pages 290–304. Springer, 2003.

[76] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. Page tables: Keeping them flat and hot (cached). *arXiv preprint arXiv:2012.05079*, 2020.

[77] Cheng-Chieh Huang, Rakesh Kumar, Marco Elver, Boris Grot, and Vijay Nagarajan. C3d: Mitigating the NUMA bottleneck via coherent DRAM caches. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[78] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. CANDY: Enabling coherent DRAM caches for multi-node systems. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[79] Chia Chen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. CAMEO: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12, 2014. ISSN: 2379-3155.

[80] Andreas Prodromou, Mitesh Meswani, Nuwan Jayasena, Gabriel Loh, and Dean M. Tullsen. MemPod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 433–444. IEEE, 2017.

[81] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. *SIGARCH Comput. Archit. News*, 31(2):84–97, May 2003.

[82] Moshe Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. Analyzing parallel programs with pin. *Computer*, 43(3):34–41, 2010.

[83] David L Dill. The mur ϕ verification system. In *International Conference on Computer Aided Verification*, pages 390–393. Springer, 1996.

[84] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer architecture news*, 41(3):475–486, 2013.

[85] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Burlingame, CA, USA, February 2015. IEEE.

[86] Ying Huang. [patch] mm,unmap: avoid flushing tlb in batch if pte is inaccessible. 2023.

[87] Ying Huang. [patch -v5 8/9] migrate_pages: batch flushing tlb. 2023.

[88] Changyeon Jo, Hyunik Kim, Hexiang Geng, and Bernhard Egger. Rackmem: A tailored caching layer for rack scale computing. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, page 467–480, New York, NY, USA, 2020. Association for Computing Machinery.

[89] Tong Xing and Antonio Barbalace. Rethinking applications' address space with cxl shared memory pools. In *Proceedings of the 4th Workshop on Heterogeneous Composable and Disaggregated Systems*, HCDS '25, page 52–59, New York, NY, USA, 2025. Association for Computing Machinery.

[90] Wentao Huang, Mo Sha, Mian Lu, Yuqiang Chen, Bingsheng He, and Kian-Lee Tan. Bandwidth Expansion via CXL: A Pathway to Accelerating In-Memory Analytical Processing.

[91] Debendra Das Sharma. Compute express link (cxl): Enabling heterogeneous data-centric computing with heterogeneous memory hierarchy. *IEEE Micro*, 43(2):99–109, 2023.

[92] Jinshu Liu, Hamid Hadian, Hanchen Xu, Huaicheng Li, and Virginia Tech. Tiered memory management beyond hotness.

[93] Zhenlin Qi, Shengan Zheng, Ying Huang, Yifeng Hui, Bowen Zhang, Linpeng Huang, and Hong Mei. Chrono: Meticulous hotness measurement and flexible page migration for memory tiering. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 835–853. ACM.

[94] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 803–817, Athens Greece, April 2024. ACM.

[95] Mark Hildebrand, Julian T. Angeles, Jason Lowe-Power, and Venkatesh Akella. A case against hardware managed dram caches for nvram based systems. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–204, 2021.

[96] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 488–504, 2021.

[97] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. {Latency-Tolerant} software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, 2015.

[98] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with {In-Network} cache coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 277–292, 2021.

[99] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.