Virtual-Memory Assisted Buffer Management In Tiered Memory

Yeasir Rayhan
Purdue University
West Lafayette, IN, USA
yrayhan@purdue.edu

Walid G. Aref
Purdue University
West Lafayette, IN, USA
aref@purdue.edu

Abstract

Tiered memory architectures have gained significant traction in the database community in recent years. In these architectures, the on-chip DRAM of the host processor is typically referred to as *local memory*, and forms the primary tier. Additional byte-addressable, cache-coherent slower memory resources, collectively referred to as *remote memory* (RMem, for short), form one or more secondary tiers. RMem is slower than local DRAM but faster than disk. In this paper, we discuss how traditional two-tier (DRAM-Disk) virtual-memory assisted buffer management techniques generalize to an n-tier setting (DRAM-RMem-Disk). We present *vmcache*^{n}, an n-tier virtual-memory-assisted buffer pool that leverages the virtual memory subsystem and OS system calls to migrate pages across memory tiers. In this setup, page migration can become a bottleneck. To address this limitation, we introduce the `move_pages2` system call that provides *vmcache*^{n} with fine-grained control over the page migration process. Experiments show that *vmcache*^{n} can achieve up to 4× higher query throughput over *vmcache* for TPC-C workloads.

ACM Reference Format:
Yeasir Rayhan and Walid G. Aref. 2026. Virtual-Memory Assisted Buffer Management In Tiered Memory . In *22nd International Workshop on Data Management on New Hardware (DaMoN '26), May 31-June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3789237.3809129

1 Introduction

Most DBMSs implement a two-tier (DRAM-Disk) buffer pool, where database pages are cached into DRAM for efficient access. Traditionally, DBMSs implement a Hash Table to index cached pages by their page identifier, i.e., PID. A PID of a cached page maps to a *virtual* memory address of the corresponding page in memory. This Hash Table-based design introduces an additional indirection as each page access requires a Hash Table Lookup. To mitigate the overhead of Hash Table-based indirection [11, 29], several alternatives have been proposed, e.g., pointer swizzling [9, 17, 18], and virtual-memory assisted buffer management [15, 21], LIPAH [19, 20]. In this paper, we focus on virtual-memory assisted buffer pools, i.e., *vmcache*. We examine how this design generalizes beyond the traditional two-tier setting and extends to an n-tier storage architecture, where the first n-1 tiers are memory-based, e.g., DRAM, Persistent Memory, CXL Memory, and the final tier is disk-based, e.g., NVMe SSD, HDD. While early work demonstrates the feasibility of tiered memory architectures in buffer pool designs [10, 21, 23, 28], the design space of virtual-memory assisted buffer management in a general n-tier setting remains largely unexplored.

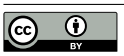
We present *vmcache*^{n}, an n-tier virtual-memory assisted buffer pool built on top of *vmcache*. *vmcache*^{n} inherits the core design principle of *vmcache*, and enforces the following invariant: *The virtual address associated with a page remains fixed throughout its lifetime.* Similar to *vmcache*, *vmcache*^{n} delegates PID translation to the OS Page Table, and retains control over page promotion and page eviction from disk to memory tiers through *libaio* interface [6]. The inclusion of additional memory tiers imposes an additional constraint on *vmcache*^{n}, thereby extending the prior invariant: *the physical frame backing a database page in memory must support dynamic mapping across different memory-resident tiers over time while ensuring that the corresponding virtual address of the page remains fixed throughout its lifetime.* In the remainder of this paper, we focus our discussion to 3-tier virtual-memory assisted buffer pools, where Tier-0, Tier-1, and Tier-2 correspond to a DRAM, a remote memory, and an NVMe disk, respectively.

2 Virtual-Memory Assisted Buffer Management in Tiered Memory

By design, in *vmcache* [15], the virtual address associated with a page remains fixed throughout its lifetime. A page may be evicted from DRAM and later re-cached at the same virtual memory address. This allows *vmcache* to safely delegate PID translation to the OS. *vmcache*^{n} extends *vmcache*'s two-tier design to an n-tier setting. The key distinction is that *vmcache*^{n}'s n-tier design requires the physical frame backing a page to be dynamically remapped across multiple memory-resident tiers without changing its virtual address. Next, we outline the design principles of *vmcache*^{n}.

1. The fundamental invariant of an n-tier virtual-memory assisted buffer pool is a stable virtual addressing along with *dynamically changeable physical frame mappings across the memory tiers.*
2. Page migration can only update the physical frame backing a PID while preserving the virtual memory address of the PID.
3. At any point in time, a page can reside in exactly one memory tier. Pages are not replicated across tiers due to the use of a single, fixed virtual address per PID.
4. Due to the invariant of stable virtual addressing, only memory tiers configured in system-RAM mode are compatible with virtual-memory assisted buffer management [3, 24].

vmcache^n is incompatible with alternative memory-tier configurations, e.g., Device Direct Access (DAX) mode [12], where memory is exposed as a character device with fixed virtual-to-physical mappings. Table 1 presents a conceptual comparison of alternative buffer pool designs with *vmcache*^{n}. *vmcache*^{n} inherits the core advantages of *vmcache* including support for variable-sized pages, graph workloads, and robust performance for both in-memory and out-of-memory workloads. In an n-tier setting, several new design



DaMoN '26, Bengaluru, India

ACM ISBN 979-8-4007-2455-8/2026/05
https://doi.org/10.1145/3789237.3809129

Table 1: Conceptual comparison of buffer management techniques in tiered memory.

	mmap	tradi. [10, 28]	Ptr swiz. [23]	Hyrise [21]	*vmcache*^{n} Sec. 2
PID-transl.	page tbl.	hash tbl.	invasive	page tbl.	page tbl.
tier-trk.	page tbl.	hash tbl.	hash tbl.	page tbl.	page tbl.
control	OS	DBMS	DBMS	DBMS	DBMS
mig-unit.	OS page	cacheline	cacheline	OS page	OS page
mig-gran.	*n*	*n*	*n*	1	*n*
data-mov.	`autonuma`	`memcpy`	`memcpy`	`mbind`	`move_pages2`
vir addr.	same	different	different	same	same
page dup.	no	yes	yes	no	no
var. size	easy	hard	hard	easy	easy
implem.	med	easy	hard	easy	easy

dimensions arise including tracking the physical location of pages (tier-trk), defining the migration unit (mig-unit) and granularity (mig-gran), specifying the data movement interface (data-mov), and handling page duplication (page dup).

2.1. Page Table Manipulation. At startup, *vmcache*^{n} reserves a virtual memory address space equaling the size of the *n*-th Tier.

Adding Pages from Disk to a Target Memory-Tier Cache. Following [21], *vmcache*^{n} uses `mbind` to place a page in a target memory tier. For example, to cache Page P3 in Tier-1, `mbind` sets the allocation policy to `MPOL_BIND`, ensuring that P3 is backed by physical memory from the target-tier bitmask (`tgt_tier`). Additionally, `MPOL_MF_MOVE` forces migration if the page already resides in another tier. When `mbind` succeeds, *vmcache*^{n} reads Page P3 from disk into the mapped physical frame using `pread`.

```
u64 offset = 3*pageSize; u64 tgt_tier = 0b0010;
int mode = MPOL_BIND; int flags = MPOL_MF_MOVE;
mbind(virtMem+offset, pageSize, mode, tgt_tier, 8*
    sizeof(tgt_tier), flags);
pread(fd, virtMem+offset, pageSize, offset);
```

Adding and Removing Pages Across Memory-Tier Caches. In an *n*-tier hierarchy, pages may migrate between two memory tiers. *vmcache*^{n} supports two system calls, namely, `mbind` and `move_pages` to migrate pages between memory tiers. Although both `mbind` and `move_pages` can migrate pages, they differ in their migration granularity. `mbind` can only migrate one page at a time.

```
u64 offset1=3*pageSize; u64 offset2=6*pageSize;
int mode = MPOL_BIND; int flags = MPOL_MF_MOVE;
u64 tgt_tier = 0b0010;
for offset in [offset1, offset2]:
    mbind(virtMem+offset, pageSize, mode, tgt
    _tier, 8*sizeof(tgt_tier), flags);
```

In contrast, *vmcache*^{n} invokes `move_pages` once to migrate multiple pages. The first argument (=0) denotes the calling process, and the second (=2) specifies the number of pages and sizes of the array arguments. The `pages` array holds virtual addresses, `tgt_tier` specifies destination tiers, and `status` reports migration outcomes. The `MPOL_MF_MOVE` flag enforces migration to the target tiers.

```
u64[] pages = {virtMem+3*pageSize, virtMem+6*
    pageSize}; int flag = MPOL_MF_MOVE;
int[] tgt_tiers={1, 1}; int[] status={-1, -1};
move_pages(0, 2, offset, tgt_tier, status, flag);
```

2.2. Page States & Synchronization. *vmcache*^{n} uses 64 bits to represent a page state. $\lceil \log_2(n-1) \rceil$ bits encode the memory-tier location, 8 bits encode the state (Unlocked, LockedShared, Locked, Marked, Evicted) and the remaining bits store a version counter yielding tier-specific states, e.g., UnlockedDRAM, UnlockedRMem.

State transition. Figure 1 shows the state transition diagram of a *vmcache*^{n} page. *vmcache*^{n} maintains tier-specific page states to allow for flexible page movements across memory tiers. Following [28], *vmcache*^{n} maintains *four* migration flags to probabilistically migrate pages between memory tiers, i.e., `Dr`, `Dw`, `Rr`, `Rw`. Assume *vmcache*^{n} accesses an Evicted Page P1 via `fix(P1)`. `fix(P1)` selects a target tier (DRAM or RMem) probabilistically using `Rr`. If DRAM is chosen, P1 transitions to the Locked state (via CAS), is read from disk, and placed in DRAM. After access, `unfix(P1)` transitions P1 to Unlocked. When DRAM utilization exceeds a threshold (e.g., 95%), *vmcache*^{n} marks unlocked pages (e.g., P1) using clock replacement. A Marked page is demoted to RMem or disk based on `Rw`. If RMem is selected, `move_pages` migrates P1 from DRAM to RMem; P1 resumes in Unlocked state. Otherwise, it is evicted to disk. For a page P2 in RMem, *vmcache*^{n} invokes `fix(P2)` and either accesses it in place or promotes it to DRAM based on `Rr`. If promoted, *vmcache*^{n} batches P2 with other Unlocked pages from RMem and migrates them together to DRAM.

2.3. Page Replacement. *vmcache*^{n} maintains a separate cache for each memory tier. Following *vmcache*, *vmcache*^{n} uses the clock replacement algorithm to evict pages from the caches when they get full. Unlike *vmcache*, in *vmcache*^{n}, there are *two* distinct eviction paths, i.e., eviction between memory tiers and eviction from a memory tier to disk. The introduction of additional memory tiers introduces additional promotion paths between two memory tiers.

Batch eviction between memory tiers. Each cache in *vmcache*^{n} maintains a separate resident set, i.e., an open-addressing hash table that indexes the resident page PIDs in the corresponding memory tier. When DRAM cache size reaches a threshold and the target tier is RMem, *vmcache*^{n} starts the following eviction routine: (1) Scan the DRAM resident set, collect the set of Marked pages selected for eviction, and lock these pages. (2) Update the location of the pages in the OS Page Table using `move_pages`. Since the pages remain memory-resident, no additional handling is required for dirty pages. (3) Remove the locked pages from the DRAM resident set and insert them into the RMem resident set. (4) Update the physical tier location metadata for each page and release the locks.

Batch promotion between memory tiers. In *vmcache*^{n}, a page may reside in memory, but in a RMem tier. When a page lookup requires accessing a page from the RMem tier, *vmcache*^{n} probabilistically decides whether the page should be promoted. If promotion is triggered, *vmcache*^{n} locks the target page, scans the corresponding resident set, and collects a set of `Unlocked` pages selected for promotion. *vmcache*^{n} locks these pages, and follows Steps 2–4 of the batch eviction routine.

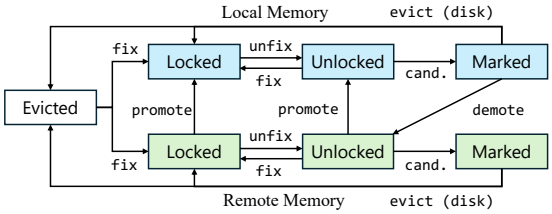


Figure 1: Page state transition diagram in a 3-tier *vmcache*^{n}.

3 MOVE_PAGES2: Efficient Page Migration Across Memory Tiers

3.1. Motivation. *vmcache*^{n} relies on the OS system calls, e.g., `mbind` [14] and `move_pages` [13], to dynamically alter the physical location of a page while preserving its virtual address. We focus on `move_pages` as it enables batched page migration. In contrast, `mbind` migrates pages individually, and does not scale.

1. TLB Shootdowns. `move_pages` migrates pages in multiple rounds, batching up to `NR_MAX_BATCHED_MIGRATION` consecutive pages per round that target the same memory tier. The batch size directly impacts performance as larger batches reduce TLB invalidation overhead. TLB shootdowns are expensive because they require inter-processor interrupts (IPIs) to invalidate TLB entries across CPUs. By default, in `move_pages`, `NR_MAX_BATCHED_MIGRATION` is set to 512. The default migration mode is `MIGRATE_SYNC`, which may trigger up to 512 TLB shootdowns per round.

2. Abort-on-failure strategy. During migration, `move_pages` invokes the kernel function `do_pages_move`, whose error-handling strategy has significant performance implications. When the kernel encounters a user-space access failure, an invalid target node, or a permission error for any page, it immediately aborts the operation. Before aborting, it migrates only the pages accumulated in the current batch, skipping all remaining pages in the input `pages` list.

3.2. Design Principle. `move_pages2` excludes the `pid` and `flags` parameters from the original `move_pages` interface and instead introduces two new parameters, `migrate_mode` and `nr_max_batched_migration`. By default, `move_pages2` sets `pid = 0` and `flags = MPOL_MF_MOVE`, thereby granting *vmcache*^{n} exclusive control over the migration process.

```
long move_pages2 (unsigned long count, void *pages[.count],
    const int nodes[.count], int status[.count],
    enum migrate_mode mode, int nr_max_batched_migration);
```

- `count`: The number of pages to process.
- `pages`: An array of page-pointers that need to be processed.
- `nodes`: An array of target NUMA node IDs.
- `status`: An array to store the migration status of each page.
- `mode`: The mode of migration.
- `nr_max_batched_migration`: The maximum number of pages that can be accumulated before TLB shootdown is invoked.

1. Reducing TLB Shootdown. `move_pages2` introduces `nr_max_batched_migration` to regulate the number of pages migrated in a single round, thereby amortizing the cost of TLB shootdown cost. In addition, `move_pages2` exposes three page migration modes through the `migration_mode` parameter.

- `MIGRATE_ASYNC` performs asynchronous migration, i.e., it proceeds with the next page even if the current page migration fails, making it a non-blocking approach.
- `MIGRATE_SYNC` performs synchronous migration, i.e., the kernel blocks until the current page migration succeeds.
- `MIGRATE_SYNC_LIGHT` does not block on page writebacks to reduce the stall time.

2. Optimistic Failure Handling. `move_pages2` adopts an optimistic error-handling strategy by allowing partial migration. Invoking a system call incurs the overhead of crossing the OS-*vmcache*^{n} boundary. `move_pages2` amortizes this overhead by migrating as many pages as possible within each invocation.

```
int migrate_pages2 (..., enum migrate_mode mode, ..., int
     nr_max_batched_migration){
again:
  nr_pages = 0; // Iterate over the pages in "from"
  list_for_each_entry_safe(folio, folio2, from, lru) {
    nr_pages += folio_nr_pages(folio);
    if (nr_pages >= nr_max_batched_migration) break;
  }
  // Batch the pages in "from" into folios and migrate
  if (nr_pages >= nr_max_batched_migration)
    list_cut_before(&folios, from, &folio2->lru);
  else
    list_splice_init(from, &folios);
  if (migration_mode == MIGRATE_SYNC)
    rc = migrate_pages_sync(..., MIGRATE_SYNC, ...);
  else
    rc = migrate_pages_batch(..., mode, ...,
          nr_max_batched_migration);
  // Store non-migrated pages in "ret_folios" for retry
  list_splice_tail_init(&folios, &ret_folios);
  if (rc < 0) {
    rc_gather = rc; list_splice_tail(&split_folios, &
     ret_folios); goto out;
  }
  rc_gather += rc;
  if (!list_empty(from)) goto again; // Prep next folios
 out:
  // Move back the non-migrated pages in the current
  // migr. round potential retry in the next round
  list_splice(&ret_folios, from);
  if (list_empty(from)) rc_gather = 0;
  return rc_gather;
}
```

Listing 1: Pseudocode of `migrate_pages2`.

```
static int do_pages_move2 (...,enum migrate_mode mode,int
     nr_max_batched_migration){
  for (i=start=0; i<nr_pages; i++) {
    if (error in copying pages[i] or nodes[i] or error in
      handling target tier)
        goto handle_error;
    if (current_node == NUMA_NO_NODE) { // Start mig.
      current_node = node; start = i;
    } else if (node != current_node) {
      // End the current migration round and migrate
      // the pages collected in the current round
      err = move_pages_and_store_status2(..., mode,
     nr_max_batched_migration);
      if (err) goto migrate_error;
      start = i; current_node = node;
    }
    // Queue page p for the current migration round
    err = add_page_for_migration();if (err > 0) continue;
    err = store_status();
    migrate_error:
      if (i + 1 == nr_pages || err < 0) {
        err1 = move_pages_and_store_status2(); // Migrate
        if (err >= 0) err = err1;
        current_node = NUMA_NO_NODE; // New mig. round
      }
      continue; // Continue migration
    handle_error:
      err1 = store_status(status, i, err, 1);
      if (err1) err = err1;
      err1 = move_pages_and_store_status2(); //Migrate
      if (err >= 0) err = err1;
      current_node = NUMA_NO_NODE; // New round@
  }
out_flush:
  if (current_node != NUMA_NO_NODE) {
    err1 = move_pages_and_store_status2(..., mode,
     nr_max_batched_migration);
    if (err >= 0) err = err1;
  }
out: return err;
}
```

Listing 2: Pseudocode of `do_pages_move2`.

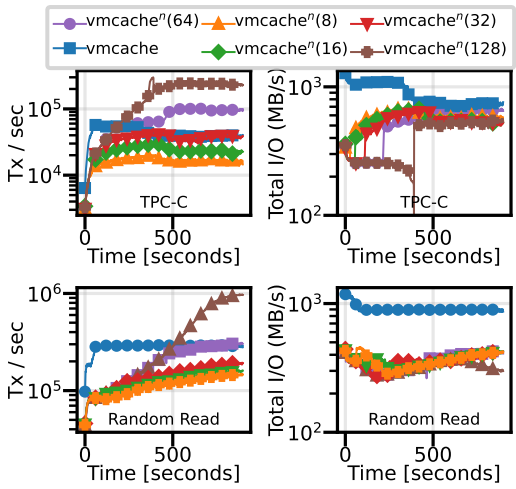


Figure 2: *vmcache*^{n} vs. *vmcache*.

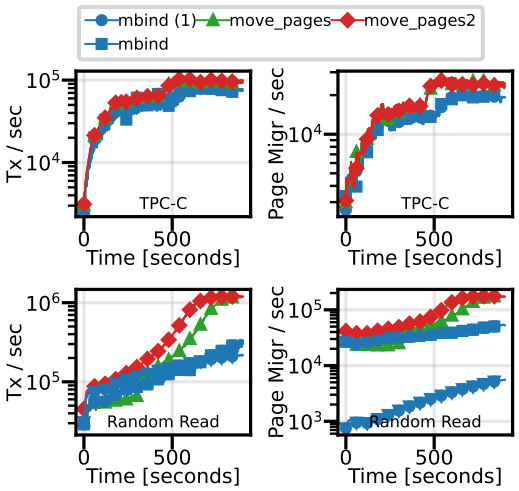


Figure 3: Performance of `move_pages2` in *vmcache*^{n}.

3.3. Implementation Details. We introduce approximately 150 lines of code changes across the following four kernel functions: `do_pages_move`, `move_pages_and_store_status`, `do_move_pages_to_node`, and `migrate_pages` to implement `move_pages2`. We implement `move_pages2` in Linux Kernel 6.8.0 on Ubuntu 24.04 LTS. The kernel disk image is available here [1].

1. Migration Modes and Bulk Batching. Our custom implementation of `migrate_pages` (cf. Listing 1) enables *vmcache*^{n} to override the default Linux migration mode, i.e., `MIGRATE_SYNC`. For `MIGRATE_SYNC`, the kernel follows the default blocking path (Line 13). Otherwise, the kernel follows the non-blocking path (Line 15). `move_pages2` further allows *vmcache*^{n} to dynamically select the maximum batch size during migration (Lines 6, 9).

2. Optimistic Error Handling. Our custom implementation of Kernel Function `do_pages_move` (cf. Listing 2) ensures that if the kernel encounters any access failures for a particular page, it does not abort immediately. Instead, it records the error in the `status` array and invokes the custom Kernel Function `move_pages_store_status2` to migrate the pages accumulated in the current round. After completing the batch, the kernel initiates a new migration round and continues processing the remaining pages in the `pages` list (Lines 24–29). `move_pages2` follows the same routine when it encounters the errors during migration (Lines 17–23).

4 Performance Evaluation

All experiments run on a CloudLab [7] sm220u node, i.e., an Intel(R) Xeon(R) Silver 4314 CPU with 2 NUMA sockets, each with 32 logical threads. We treat DRAM of NUMA Socket 0 as local memory (Tier-1), DRAM of NUMA Socket 1 as remote memory (Tier-2) and a 960 GB Samsung PCIe4 NVMe as disk (Tier-3). All experiments use 32 threads of NUMA Socket 0. Following [15], we use the same TPC-C and a key-value workload that consists of random point lookups. The key–value workload uses 8-byte uniformly distributed keys and 120-byte values. The TPC-C and random-read workloads span approximately 190 GB and 130 GB, respectively.

***vmcache*^{n} vs. *vmcache*.** Figure 2 compares the performance of *vmcache*^{n} and *vmcache* over time for the TPC-C and random-read workloads, with the migration flags set to 1 and 0.1, respectively. The local memory capacity is fixed at 32 GB, while the remote memory capacity varies from 8 GB to 128 GB. At time T_0, the remote tier is empty and fills over time. For TPC-C, demotions first go to remote. Once it reaches 95% capacity, excess spills to SSD, causing bursty writebacks driving the I/O spikes. For TPC-C, *vmcache*^{n} achieves up to 1.67× and 3.82× higher throughput than *vmcache* when the remote memory capacity is 2× and 4× the DRAM capacity, respectively. Although *vmcache*^{n} reduces disk I/O, when the remote memory is small, traffic between memory tiers becomes the performance bottleneck. For the random-read workload, with no writes, the performance gain is not significant, e.g., *vmcache*^{n} achieves 1.36× better transaction throughput when remote memory is 4× the DRAM capacity.

Performance evaluation of `move_pages2`. Figure 3 gives the performance of `move_pages2` in *vmcache*^{n} against baseline page migration system calls, i.e., `mbind` and `move_pages`. `mbind(1)` sets the batch eviction size between memory tiers to 1. For the rest, the batch eviction size is set to 512. `move_pages2` exhibits similar performance to `move_pages` for the TPC-C workload. However, for the random-read workload, `move_pages2` achieves 1.42×, 1.32× better query and page migration throughput over `move_pages`, respectively, due to the maximum number of memory pages `move_pages2` can batch together in a single round compared to the baselines, thus amortizing the cost of TLB invalidation.

5 Concluding Remarks

We present *vmcache*^{n}, an *n*-tier virtual-memory assisted buffer pool that achieves up to 4× higher TPC-C throughput than *vmcache*. For virtual-memory assisted buffer pools, remote memory has a cost break-even point when its capacity is between 1× and 2× DRAM capacity. Below this, page migration overheads between memory tiers outweigh any capacity gains, resulting in negative QPS/$. `move_pages2` mitigates this overhead through batched migrations, but overheads still persist. Notably, less than 0.005% of time is spent in kernel-user transitions [26], indicating that microkernel-based designs are unlikely to help [16]. Rather, more efficient page migration implementations are likely to yield greater benefit, be it in user space [22] or kernel space [2, 4, 5, 8, 25, 27].

References

[1] 2026. Linux Disk Image with `move_pages2` (Kernel 6.8.12). urn:publicid:IDN+utah.cloudlab.us+image+pmoss-PG0:LINUX6.8.12_MIGRATE.

[2] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *USENIX*. 27–39.

[3] Ramesh Aravind and Groves John. 2024. Study of CXL Memory Sharing with FamFS and its Use cases. In *HiPC Workshops*. IEEE, 73–77. doi:10.1109/HIPCW63042.2024.00022

[4] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H. Loh. 2017. Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries. In *PACT*. 273–287.

[5] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient Address Translation for Architectures with Multiple Page Sizes. In *ASPLOS*. 435–448.

[6] Daniel Ehrenberg. 2026. The Asynchronous Input/Output (AIO) interface. https: https://github.com/littledan/linux-aio. Accessed February 11, 2026.

[7] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *USENIX*. USENIX Association, 1–14.

[8] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. 2016. Range Translations for Fast Virtual Memory. *IEEE Micro* 36, 3 (2016), 118–126.

[9] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi A. Kuno, Joseph A. Tucek, Mark Lillibridge, and Alistair C. Veitch. 2014. In-Memory Performance for Big Data. *Proc. VLDB Endow.* 8, 1 (2014), 37–48. doi:10.14778/2735461.2735465

[10] Xiangpeng Hao, Xinjing Zhou, Xiangyao Yu, and Michael Stonebraker. 2024. Towards Buffer Management with Tiered Main Memory. *SIGMOD* 2, 1 (2024), 31:1–31:26.

[11] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the looking glass, and what we found there. In *SIGMOD*. 981–992. doi:10.1145/1376616.1376713

[12] Intel Corporation. 2024. CXL* Type 3 Memory Device Software Guide. Revision 1.1. https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://cdrdv2-public.intel.com/643805/643805_CXL_Memory_Device_SW_Guide_Rev1_1.pdf. Accessed February 9, 2026.

[13] Michael Kerrisk. 2025. move_pages(2) — Linux manual page. https://man7.org/linux/man-pages/man2/move_pages.2.html. Accessed: February 11, 2026.

[14] Andi Kleen. 2026. mbind — Linux manual page. https://man7.org/linux/man-pages/man2/mbind.2.html. Accessed: February 11, 2026.

[15] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proc. ACM Manag. Data* 1, 1 (2023), 7:1–7:25. doi:10.1145/3588687

[16] Viktor Leis and Christian Dietrich. 2024. Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware. *VLDB* 17, 8 (2024), 2115–2122. doi:10.14778/3659437.3659462

[17] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. 185–196. doi:10.1109/ICDE.2018.00026

[18] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. https://vldb.org/cidrdb/2020/umbra-a-disk-based-system-with-in-memory-performance.html

[19] Riki Otaki, Charles Benello, Aaron J. Elmore, and Goetz Graefe. 2025. Resource-Adaptive Query Execution with Paged Memory Management. In *CIDR*. https://vldb.org/cidrdb/2025/resource-adaptive-query-execution-with-paged-memory-management.html

[20] Riki Otaki, Jun Hyuk Chang, Aaron J. Elmore, and Goetz Graefe. 2025. Enhancing Transaction Processing through Indirection Skipping. *Proc. VLDB Endow.* 18, 11 (2025), 4104–4116. doi:10.14778/3749646.3749680

[21] Niklas Riekenbrauck, Marcel Weisgut, Daniel Lindner, and Tilmann Rabl. 2024. A Three-Tier Buffer Manager Integrating CXL Device Memory for Database Systems. In *HardBD & Active*. 395–401.

[22] Felix Schuhknecht and Nick Rassau. 2026. Taking the Leap: Efficient and Reliable Fine-Grained NUMA Migration in User-space. *CoRR* abs/2602.05540 (2026). arXiv:2602.05540 http://arxiv.org/abs/2602.05540

[23] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *SIGMOD*. 1541–1555. doi:10.1145/3183713.3196897

[24] Marcel Weisgut, Daniel Ritter, Florian Schmeller, Pınar Tözün, and Tilmann Rabl. 2025. CXL-Bench: Benchmarking Shared CXL Memory Access. In *ADMS*. https://pure.itu.dk/en/publications/cxl-bench-benchmarking-shared-cxl-memory-access/

[25] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *USENIX*. 19–35.

[26] Tong Xing and Antonio Barbalace. 2025. Rethinking Applications' Address Space with CXL Shared Memory Pools. In *HCDS*. 52–59. doi:10.1145/3723851.3723858

[27] Zi Yan, Daniel Lustig, David W. Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *ASPLOS*. 331–345.

[28] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David E. Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *SIGMOD*. 2195–2207. doi:10.1145/3448016.3452819

[29] Xinjing Zhou, Viktor Leis, Xiangyao Yu, and Michael Stonebraker. 2025. OLTP Through the Looking Glass 16 Years Later: Communication is the New Bottleneck. In *CIDR*. https://vldb.org/cidrdb/2025/oltp-through-the-looking-glass-16-years-later-communication-is-the-new-bottleneck.html