The Kernel's Write: Application Read-Only Memory

Hui Sub Shim
Stanford University
Stanford, California, USA
hsshim@stanford.edu

Katherine Mohr
Stanford University
Stanford, California, USA
kmohr@cs.stanford.edu

Philip Levis
Stanford University
Stanford, California, USA
pal@cs.stanford.edu

Abstract

Alongside power, DRAM has become a major limiting factor in datacenter growth. As DRAM's cost-per-bit scaling has slowed over the past decade, a class of emerging memory technologies, called Long-term RAM (LtRAM), offers a path to denser and cheaper main memory.

However, LtRAM has three main drawbacks: asymmetric read/write latencies, limited endurance, and coarse write granularity. In an attempt to isolate software from these drawbacks, LtRAM technologies such as Intel Optane copy an approach from flash devices, introducing a translation layer that manages wear-leveling, address remapping, and read/write caching. Prior experimental studies have found these operations add significantly to LtRAM latency.

Rather than making LtRAM look like DRAM, we propose redesigning the hardware/software interface to offload more responsibility to the operating system. This design hinges on one central property, Application Read-Only Memory (AROM): LtRAM pages are read-only to applications and written only by the OS during page migrations. AROM is enforced by leveraging copy-on-write (CoW): application writes to LtRAM trigger a fault that migrates the page back to DRAM before the store is applied. This invariant allows LtRAM management to shift from the on-DIMM controller to the operating system, drastically simplifying the DIMM's hardware. With this approach, we aim to match the performance of pure DRAM on read-mostly workloads while delivering LtRAM's density and cost advantages.

CCS Concepts: • **Software and its engineering** → **Memory management**; • **Hardware** → *Non-volatile memory*; Emerging technologies.

Keywords: Long-term RAM, Application Read-Only Memory, memory management, copy-on-write, wear leveling

ACM Reference Format:
Hui Sub Shim, Katherine Mohr, and Philip Levis. 2026. The Kernel's Write: Application Read-Only Memory. In *4th Workshop on Disruptive Memory Systems (DIMES '26), September 29–October 2, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3831664.3837780

DIMES '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2880-8/26/09
https://doi.org/10.1145/3831664.3837780

1 Introduction

Memory has become the dominant share of server cost. Microsoft Azure reports that DRAM accounts for over half of total server cost [18, 23], a problem that has accumulated as DRAM cost-per-bit scaling has slowed [19].

Long-term RAM is a recent proposal to mitigate this issue. LtRAM is a class of emerging memory technologies that compete with DRAM's read latency, but with lower per-bit cost. Some examples include RRAM [1, 31], PCM [15], FeFET [7], and MRAM [3, 11]. Prior work has identified LtRAM as well-suited for long-lived, read-mostly data [19]. Replacing the DRAM that holds such data with LtRAM has the potential to reduce DRAM footprint and overall main memory cost.

However, LtRAM has key limitations: read/write latencies are asymmetric, endurance is limited, and write granularity is coarse. Intel Optane, a discontinued phase-change memory (PCM) module, is the canonical commercial example of LtRAM. To minimize software-level changes, Optane adopted the translation-layer pattern from solid state drives (SSD), pushing wear-leveling, address remapping, and buffer caching into on-DIMM hardware. This hardware complexity added latency and inflated module cost, diminishing the underlying technology's performance and cost advantages.

We argue that forcing DRAM compatibility on LtRAM adds too much complexity, eroding the performance and cost advantages LtRAM can deliver. A new interface between the OS and memory would let the OS manage LtRAM properties with metadata it already has, rather than hiding them in the module. With this new interface, we propose a mixed DRAM and LtRAM main memory that aims to match pure-DRAM performance while reducing per-server DRAM footprint and overall main memory cost. Our latency projection places LtRAM 26–79% faster than Optane and between 9% faster and 220% slower than DRAM on per-access read latency, while providing density and cost benefits.

This paper makes four contributions:

1. An analysis of the performance hits incurred by Intel Optane's on-module translation layer (§3.2).
2. The **Application Read-Only Memory (AROM) invariant**: only the OS writes to LtRAM, enforced by copy-on-write (CoW) on application stores (§4).

3. An interface that uses AROM to move LtRAM management from the on-DIMM controller to the OS, avoiding Optane's controller overheads (§4).
4. A prototype design on the Enzian [4] platform with NOR flash as the LtRAM (§5).

2 Motivation: The Problem with DRAM

DRAM's share of server cost has grown for two reasons. Memory demand has increased with core counts and per-server working-set sizes, while DRAM cost-per-bit scaling has slowed for over a decade. DRAM cell density may continue to improve through emerging technologies such as 3D-stacked DRAM [9] and vertical-channel access transistors [8]. However, manufacturing constraints on capacitor scaling limit how quickly cost-per-bit can fall [19]. The 2025–2026 demand surge driven by AI has exacerbated the cost pressure, with prices for mainstream DIMMs roughly doubling in 1Q26 alone [6, 27].

The DRAM price spike from the AI demand surge may resolve as suppliers expand capacity, but the underlying cost-per-bit slowdown will not. This slowdown reflects fundamental technological limits, not market dynamics. DRAM's share of server cost is therefore projected to keep growing [13].

To address this unsustainable growth, software must use DRAM more efficiently, or hardware must adopt cheaper memory alternatives. Software techniques such as memory compression [14] and page deduplication slow DRAM usage growth but cannot lower the cost-per-bit. Memory tiering [20, 23, 28] demotes cold pages to slower, cheaper hardware alternatives such as Compute Express Link (CXL) [33], reusing previous-generation DRAM or new memory devices, but it applies only to cold pages, and both reads and writes to the slower tier incur a latency penalty. The next section explores an alternative approach: LtRAM.

3 Long-term RAM (LtRAM)

Long-term RAM (LtRAM) is a class of memory technologies intended as a denser, cheaper complement to DRAM [19]. LtRAM matches DRAM's read latency and reaches densities several times those of current DRAM, with associated reductions in cost-per-bit. However, this density comes with tradeoffs: writes are slower, write granularity is coarser than a cache line, and endurance is bounded.

Given these write restrictions, LtRAM is best suited for long-lived, read-mostly data: code pages, machine learning inference weights [17], and in-memory stores such as Redis and Memcached [19]. Substituting LtRAM for the DRAM that holds such data can lower per-server memory cost.

To make this concrete, we survey the candidate LtRAM technologies and the design space they expose (§3.1) and analyze how Intel Optane's controller mechanisms sacrificed those gains in practice (§3.2).

Device	Read lat.	Write lat.	Endurance	Write gran.	Cost per bit
DRAM	80 ns	80 ns	∞	cache line	high
3D V-RRAM	100 ns	1 μs	10^6	page	low
3D FeFET	200 ns	10 μs	10^5	page	low
PCM	300 ns	1 μs	10^8	256 B	mid
STT-MRAM	20 ns	20 ns	10^{15}	word	high

Table 1. Representative LtRAM memories and their characteristic shape, with DRAM as reference. Numbers should be read for shape rather than precision, as each row can shift by orders of magnitude with material and circuit choices [22]

3.1 The LtRAM design space

Table 1 summarizes four LtRAM technology candidates [1, 3, 7, 11, 15]. Although the candidates differ in physical mechanism, they generally share the same shape: large read/write latency asymmetry, coarse write granularity, and hard endurance limits. LtRAM is the abstraction over this shape. These memory technologies can continue density and cost-per-bit scaling where DRAM has slowed [19].

3.2 Case Study: Intel Optane

Intel Optane, the only LtRAM-class memory ever available as a DIMM, illustrates what goes wrong when non-DRAM memory is hidden behind a DRAM interface.

Each of Optane's controller-side mechanisms was introduced to hide memory complexity from software. Hiding that complexity comes at a measurable cost: a read-modify-write on every sub-256 B store, an Address Indirection Table (AIT) lookup on every read, a 160× tail-latency spike from internal wear-leveling migrations, and a bandwidth collapse under mixed read/write workloads. Table 2 summarizes four of Optane's hardware mechanisms and their resulting overheads. After a brief overview of Optane, the following subsections address each of its hardware overheads in turn.

3.2.1 Intel Optane Background. Intel Optane DC Persistent Memory Module (DCPMM) is a byte-addressable non-volatile DIMM that sits on the memory bus alongside DRAM and exposes 3D-XPoint media to the CPU at cache-line granularity. The on-DIMM controller (XPController) mediates between cache-line granularity transactions from the CPU and the coarser-granularity 3D-XPoint media, shown in Figure 1. Three structures drive Optane's access latency:

3D-XPoint media. 3D-XPoint is the phase-change memory (PCM) media that backed Optane. It has a 256 B access granularity, which Intel calls an XPLine.

AIT and cache. The controller maintains the AIT, a host-invisible map from CPU physical addresses to media locations. This map implements hardware wear-leveling and bad-block remapping. The full AIT lives in the media, and a 16 MB on-DIMM DRAM buffer caches recent entries at 4 KB granularity [12, 29, 30].

Optane Mechanism	Resulting Overhead	LtRAM Interface Fix
Cache-line Granularity Mismatch (§3.2.2)	75% write throughput drop, 2× random-read latency	Cache-line reads, page writes; OS page management naturally coalesces writes
Wear-Leveling (§3.2.3)	Opaque migration fires every ~3.4 MB written; ~60 μs tail read latency per event (>160× latency)	OS-side wear-leveling; hardware has no migration path
Address Indirection Table (§3.2.4)	76–200 ns added to every read	OS-side wear-leveling; no on-module map
Workload-Agnostic Design (§3.2.5)	70% read bandwidth drop under 50/50 R/W; 4× more sensitive to mixed workload than DRAM	Read-mostly by design; OS-paced writes via token allocator

Table 2. Each Optane mechanism carries an overhead that the LtRAM interface eliminates by shifting responsibility to the OS.

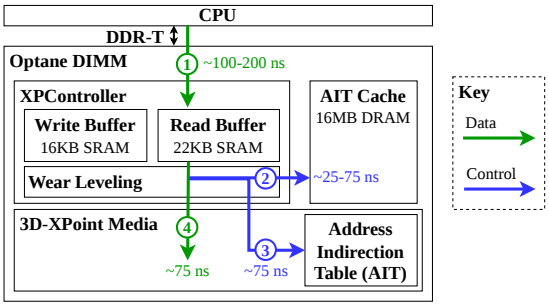

Figure 1. Optane internals, with a sample read path. Worst-case reads traverse the read buffer, AIT cache, and media-resident AIT before fetching data from the 3D-XPoint media. Numbers are approximate based on prior work [21, 29].

Write and Read buffers. The DIMM controller maintains a small write-combining buffer (~12–16 KB) that coalesces sub-XPLine writes and a read buffer (~16–22 KB) that caches XPLines to speed up sequential reads [29, 30].

These components introduce additional complexity to Optane's hardware, increasing manufacturing costs and making Optane's performance highly sensitive to workload.

3.2.2 Cache-line Granularity Mismatch.

While modern processors have 64 B cache lines, the 3D-XPoint memory technology naturally handles reads and writes in 256 B XPLines. Consequently, random cache-line writes suffer, as sub-XPLine writes must perform a read-modify-write (RMW) operation, reading the XPLine from media, modifying it, and writing it back. While sequential writes can be coalesced in a write-combining buffer, random 64 B writes incur a 4× write amplification, dropping throughput from ~2.3 GB/s to ~0.56 GB/s (~75% degradation) [29].

Random cache-line reads face the same penalty. The controller hides read latency of sequential 64 B reads by fetching whole XPLines into an on-DIMM read buffer, achieving ~169 ns sequential read latency. Meanwhile, random reads incur 4× amplification and ~305–374 ns latency [10, 32].

3.2.3 Wear-Leveling.

Optane's controller performs wear-leveling internally. When writes concentrate on a small region, the controller migrates data off the worn cells. Wang et al. [29] measure that XPLine overwrites on Optane trigger such a migration roughly every 14,000 writes (about 3.4 MB). Tail latencies reach roughly 60 μs per read request, over 160× the typical random-read latency.

3.2.4 Address Indirection Table.

Intel Optane uses an AIT and corresponding cache for wear-leveling and bad-block management. Every read consults the AIT before reaching the media. Liu et al. [21] measure reads with AIT-cache hits at 351 ns and misses at 427 ns. The 76 ns gap is the extra media access required on a miss to fetch the AIT entry. On a 128 GB DIMM, the 16 MB AIT cache covers only ~8 GB of address space, or ~6% of the device (assuming standard cache metadata: valid, tag, LRU, dirty, ECC). Therefore, 94% of random accesses miss the AIT cache and pay the 427 ns worst-case path [12, 21, 29, 30].

3.2.5 Workload-Agnostic Design.

Optane was designed as a memory-bus-attached DIMM, intended to support the same read/write workload mix as DRAM. Microbenchmark measurements show that DRAM sustains its bandwidth across all read/write mixes, while Optane does not. Izraelevitz et al. [12] measure that Optane's aggregate sequential bandwidth collapses from ~30 GB/s on pure reads to ~10 GB/s under a 50/50 mixed workload, a 67% loss in read throughput.

Optane relied on complex hardware logic to present a simple, DRAM-compatible interface to software, but each of these mechanisms added measurable performance overheads. Yet the 3D-XPoint media was not the main source of these costs; the DRAM-compatible contract was. Generalizing to LtRAM, the bottleneck is not the memory technology but the hardware/software interface: whenever the controller must hide granularity mismatch, wear leveling, address indirection, and mixed read/write behavior, LtRAM inherits the same overheads that burdened Optane. The limitation is therefore one of interface design rather than memory technology, and the next section addresses how to redefine the interface so the system can exploit LtRAM's strengths.

4 Proposed Hardware/Software Interface

We introduce a HW/SW interface that enforces an invariant we call **Application Read-Only Memory (AROM)**. Under AROM, LtRAM is read-only to applications, and the kernel may only write to LtRAM when migrating pages from DRAM. This design relocates LtRAM management from the

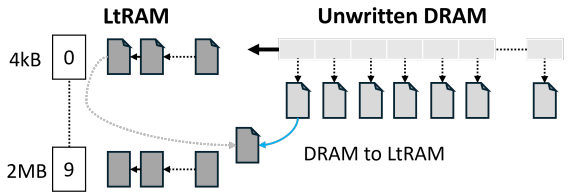

Figure 2. DRAM to LtRAM Migration. LtRAM is a distinct memory zone with its own free list. Pages exceeding a write-recency threshold are migrated from DRAM to LtRAM via Linux's existing page management.

on-DIMM controller to the OS. The device-side hardware exposes reads at cache-line granularity, writes at 4 KB page granularity, and cell health metadata; the OS manages all policy decisions with its existing structures.

4.1 Controller Requirements

The device-side controller has three responsibilities:

1. accept reads at cache-line granularity,
2. accept writes at 4 KB page granularity, and
3. expose each memory block's health to the OS.

Reads remain at cache-line granularity, where LtRAM technologies achieve read latency competitive with DRAM. The write granularity is set to 4 KB, a multiple of every LtRAM technology's write granularity (Table 1). Writes are therefore aligned to the memory's boundary, structurally eliminating the read-modify-write path that drove Optane's 75% write-throughput collapse (§3.2.2). The controller reports per-block health to the OS, which uses this to seed its wear-leveling state with each block's current wear level.

The controller does not implement wear-leveling. This eliminates the AIT and its overheads: the AIT cache lookup and the ~75 ns media access incurred on the 94% of random reads that miss the AIT cache (§3.2.4). It also removes the ~60 μs tail-latency spikes from hardware-driven wear-leveling migrations (§3.2.3). Only the memory's intrinsic access latency remains, surfaced through a simpler controller without an AIT or its AIT DRAM cache.

4.2 OS Requirements

The operating system provides four guarantees:

1. the only writes issued to LtRAM are 4 KB page writes from the kernel migration path;
2. application stores to an LtRAM-resident page trigger a copy-on-write (CoW) fault that migrates the page back to DRAM before the store applies;
3. pages migrated into LtRAM are read-mostly, avoiding the mixed-workload bandwidth collapse (§3.2.5); and
4. write traffic to LtRAM is balanced across cells, replacing the controller's wear-leveling (§3.2.3).

Guarantees (1) and (2) enforce AROM. LtRAM is read-only from the application's perspective; only the kernel migration thread writes to it, and only at 4 KB granularity. Aligning to 4 KB lets the OS manage LtRAM through existing page mechanisms. Application traffic also never reaches the media at sub-page granularity, so the controller never incurs the read-modify-write path. Guarantees (3) and (4) move migration policy and wear leveling from the controller to the OS: page migrations are handled by a kernel policy that classifies pages by write frequency, and endurance is managed by the OS at a page granularity (e.g., via a token allocator). §5 describes possible implementations of both.

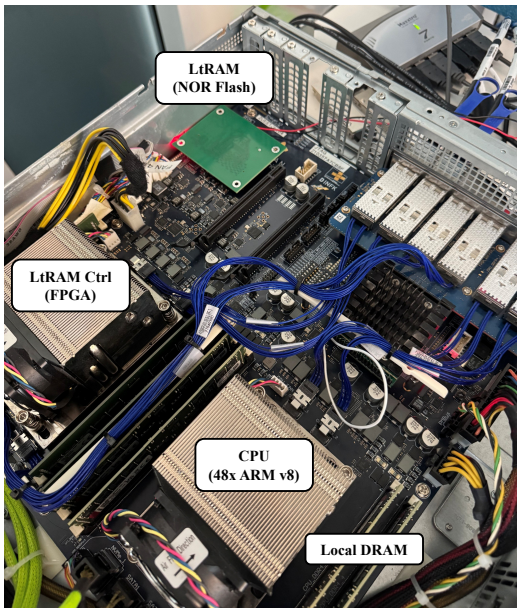

Figure 3. The Enzian research platform [4] with NOR flash LtRAM device attached for testing.

5 Implementation

We prototype the LtRAM interface on the Enzian platform [4]: a server-class ARM CPU paired with a coherent FPGA fabric, configured so the FPGA acts as the LtRAM memory controller and exposes LtRAM to the CPU through Enzian's cache-coherent interconnect. The CPU runs Linux kernel 6.8 with our page-migration extensions for LtRAM.

We chose NOR flash for the prototype for its availability and low cost: a 256 MB Micron Serial NOR Flash device [24] (Figure 3) drops onto the FPGA board, whereas an MRAM or RRAM prototype would have required a research-specific tape-out. NOR is not a canonical LtRAM technology, but it shares the asymmetry shape the interface targets: byte-addressable reads, 256 B writes, 4 KB erases, and 10^5 erase cycles per block. Measured chip-level latencies are 500 ns for a 128 B read (Enzian's cache line), 80 μs for a 256 B write, and 18 ms for a 4 KB erase.

5.1 Controller

The FPGA-based memory controller implements the three responsibilities specified in §4.1. The 4 KB write granularity coincides with both the OS base page size and the NOR device's erase block size, eliminating controller-side merging.

Beyond exposing the interface, the controller adds two optimizations that preserve read performance under the read-mostly workloads LtRAM serves. First, incoming writes are absorbed into an SRAM buffer and acknowledged before the write is committed to NOR flash, so reads to other memory regions are not stalled by the slower device-level program. Second, the erase command is exposed to the OS rather than triggered internally, so the OS can schedule erases during periods of low read pressure. Everything else, including wear distribution and placement, lives in the OS.

Removing the AIT eliminates its on-DIMM DRAM cache, the media region reserved for AIT storage, and the lookup latency on the read critical path (§3.2.4). OS scheduling of erases and migrations further avoids the tail-latency from hardware wear-leveling (§3.2.3). The controller retains only a small bad-block remap table whose coarse granularity (e.g., 64 KB per entry in Optane) lets it reside entirely in SRAM.

5.2 Operating System

The OS implementation realizes the four guarantees of §4.2 through two mechanisms: memory-aware page placement and a token-based wear-leveling subsystem. They keep LtRAM writes within the device's endurance budget and scale to multi-terabyte capacities without per-page metadata bloat.

5.2.1 Page Placement Policies. The OS implements three placement policies: allocation, migration to LtRAM, and migration to DRAM.

Allocation. LtRAM is implemented as a new Linux zone with its own free-page list; the allocator selects DRAM or LtRAM per request based on write intensity. Pages are placed directly on LtRAM only when the kernel or application can establish that the data is read-only by construction (executable code, shared libraries, read-only memory-mapped files). Everything else is first allocated in DRAM and may move to LtRAM later via the migration policy.

Migration to DRAM. All LtRAM pages are write-protected. On a write fault, the kernel performs a CoW migration: it allocates a DRAM page, copies the LtRAM page's contents, updates the page table, and executes the store. The original LtRAM page is then freed and its underlying block scheduled for erase when the device is idle. All application-level writes execute on DRAM, preserving the AROM invariant.

While a successful page migration policy would keep write-heavy pages off LtRAM, a workload phase change could still flip a large LtRAM region from read-mostly to write-heavy all at once: every store triggers a CoW fault and a migration to DRAM, which can exhaust the DRAM free list. Unlike CXL tiering [20, 23, 28], where misplacement only slows reads, misplacement in LtRAM pays a per-store copy cost on the critical path. Standard back-pressure—admission control on migration and DRAM-occupancy watermarks that throttle placement before DRAM fills—can prevent this during write-heavy periods.

Migration to LtRAM. Only read-mostly pages should be migrated to LtRAM. To determine which pages have not been written recently, we reuse existing page dirty bit tracking mechanisms in the kernel. On every scan interval T, the kernel scans the target process's PTEs: pages with a clear dirty bit become migration candidates, and the rest have their dirty bits cleared for the next scan. Candidates are migrated to LtRAM in scan order, subject to the endurance token budget (§5.2.2).

This approach is cheap but imprecise, and developing a migration policy that balances efficiency and accuracy is left as an open research question. LtRAM migration decisions are based on whether a page is *read-mostly*, not purely cold, so migration policies from existing software-only tiering work [14, 16, 20, 23, 25, 28] may not translate over. Our dirty-bit-based classifier is the simplest and lowest-overhead option, but thresholding alone can be too coarse: a page that passes the threshold in one observation window may be written in the next. A small online predictor could adapt to workload patterns from a few features (e.g., time since last write, recent write frequency), but adds tuning overhead. The right choice depends on the workload's diversity, the available per-page metadata budget, and how often workloads change at runtime.

5.2.2 Wear-leveling by token allocation. Our token allocator releases tokens at a rate determined by the device's endurance budget and target deployment lifetime. For an LtRAM with N pages rated for E erases each over a T-second deployment, the rate is $r = NE/T$ tokens per second. Every migration to LtRAM must consume a token; the kernel defers a migration if no tokens are available. Tokens may be banked across idle windows, so an under-used workload can trade idle time for a later migration burst while the long-run average stays within budget by construction. Because the OS is the only writer and the token allocator paces those writes, the device meets its lifetime budget by construction without persisting per-block erase counts; only the running token balance is required. This bounds total wear but does not enforce uniformity across the chip's physical sections. Achieving uniform wear in this paradigm is left as future work and discussed in Sec 7.

When there are more migration candidates than the budget permits, the kernel defers the surplus to the next interval. Such deferral is benign to application performance and even beneficial for placement accuracy, since a page that stays read-mostly across further intervals is a safer LtRAM target than one migrated early.

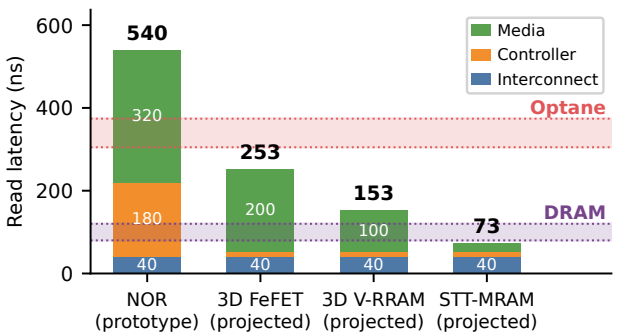

Figure 4. Projected read latency through proposed interface on a DDR-class interconnect, decomposed into interconnect, on-device controller, and media access. DRAM (80–120 ns) and Optane (305–374 ns) bands serve as reference points.

Under write-heavy workloads, fewer pages qualify as read-mostly, so LtRAM is less utilized and the system behaves like a conventional DRAM-only system, swapping colder pages to lower-tier storage to free DRAM for the working set when DRAM is exhausted.

5.2.3 What changes in the kernel? Our design reuses much of the page management machinery already present in the Linux kernel. Adding a new zone at the top of the zone fallback order separates the DRAM and LtRAM allocation paths and prevents allocations onto LtRAM unless explicitly requested, keeping write-heavy pages off LtRAM. The CoW fast path already handles fault-on-write semantics; we extend the trigger so LtRAM-resident pages are marked CoW by default and write faults migrate the contents to DRAM before modification. Migration to LtRAM requires the largest modifications, as it involves determining which pages should be migrated to LtRAM and implementing a token allocator.

5.2.4 What application-level changes must be made? AROM is transparent to applications: a process sees only its virtual address space and never learns whether a given page is backed by DRAM or LtRAM. The kernel owns placement, migrating read-mostly pages to LtRAM and faulting them back to DRAM on write, so no program must be rewritten or tuned to benefit. Whether a richer hint interface would pay off is an open design question.

5.3 Projected Read Latency

Figure 4 projects end-to-end read latency for a 128 B cache-line access through our HW/SW interface on a DDR-class interconnect. Media latencies come from Table 1. The three projected technologies take representative per-device read latencies: 200 ns for 3D FeFET [7], 100 ns for 3D V-RRAM [1], and 20 ns for STT-MRAM [3]. The interconnect is estimated at 40 ns, the residual of DRAM's 80 ns budget after subtracting a 30 ns cell access [15] and a 10 ns on-DIMM controller.

The on-device controller is based on our NOR prototype; we measure 500 ns for controller and media combined. Media access accounts for 320 ns, and the remaining 180 ns is the controller, or 36 cycles at NOR's 200 MHz clock. Reclocked to a DDR5-5600 clock (2.8 GHz), the same 36-cycle budget drops to 13 ns, which we adopt for the projected technologies. This estimate is conservative: 16 of the 36 cycles are wait states intrinsic to off-the-shelf NOR's command pipeline that a purpose-built LtRAM controller would not incur. The interconnect is held constant at 40 ns across all technologies, assuming every candidate would sit on the same DDR bus.

All LtRAM technologies improve on Optane by 26–79%, placing them closer to DRAM. Against DRAM's 80 ns baseline, the NOR prototype projects to 6.8×, 3D FeFET to 3.2×, 3D V-RRAM to 1.9×, and STT-MRAM to 0.91× (about 9% faster than DRAM). Although most projections are slower than DRAM's read latency, they are far faster than the next tier of the memory hierarchy, making LtRAM a competitive complement to DRAM as main memory. The projections characterize single-access latency in isolation; at the application level, cache, prefetching, and memory-level parallelism overlap per-access costs, so workload-level performance on read-mostly data may approach DRAM more closely than the per-access numbers suggest. Application-level benchmark evaluation remains future work.

6 Related Work

DRAM-compatible NVM modules. Intel Optane is the canonical example: a DRAM-compatible random-access interface backed by an on-DIMM translation layer [30]. As §3.2 showed, the translation layer inflates latency and module cost, eroding the cost advantage that made the memory device interesting. The shortcoming lies not in the underlying memory technology but in the interface.

Software-only tiering. Google's software-defined far memory built on zswap [14] shares the closest approach: OS manages a slower memory tier (compressed RAM) with the cost deferred to access time (decompression at fault). TPP [23], Memtis [16], Colloid [28], and Alto [20] extend the same OS-managed-tier pattern to CXL-attached memory, but none reason about endurance, and all position the second tier as a slow tier between DRAM and storage. Our design extends zswap's approach to a HW/SW co-design problem: LtRAM serves alongside DRAM as main memory rather than as a slow tier, and the deferred cost is a CoW migration paid only on writes to LtRAM pages, rather than decompression on both reads and writes.

HW/SW co-design framings. Microsoft's Managed Retention Memory work [17] argues that the right abstraction for emerging memory is neither storage class memory nor a DRAM replacement, but a category with its own interface. Li et al. [19] laid out a similar split between long-term and

short-term RAM. Pond [18] pushes in a related direction at the level of CXL pooling. This paper builds on those framings and proposes a concrete HW/SW interface for LtRAM.

7 Discussion

LtRAM technologies provide density and cost benefits, but historically have incurred high performance overheads from the memory controller, thus limiting adoption. Redesigning the HW/SW interface to move LtRAM management into the operating system avoids the need for complex controller hardware, but leaves open questions about OS-level LtRAM management. Prior software-only tiering work (§6) provides potential answers to these questions, but it does not account for the specific read/write asymmetry and endurance-limited nature of LtRAM. In this section, we explore three challenges introduced by this interface: wear-leveling read-only pages, handling huge pages, and giving applications finer control over data placement.

Shifting wear leveling from the DIMM to the operating system reduces hardware complexity, but raises new questions about how to integrate it into existing kernel structures. Free-page lists alone may be insufficient because read-only pages such as code binaries and machine-learning weights can remain pinned to the same LtRAM blocks for long periods, forcing the more frequently written blocks to absorb disproportionate wear. Similarly, pages that the operating system treats as unmovable, such as kernel code pages, may remain fixed indefinitely and never participate in wear-leveling rotation. Page-cache pages present a related challenge, as clean cached data may remain resident without being modified for long periods. As such, the operating system may need to perform more explicit wear leveling than occurs naturally through page reclamation, LRU management, and free-page allocation; however, it is unclear how best to achieve uniform wear when managing read-only and unmovable pages. One solution is to evict pages from the least-worn blocks when the wear gap exceeds a threshold, but open questions remain: when to trigger eviction, what threshold to use, and how to pick candidates without inflating overhead.

Huge pages represent another open question. The operating system today maps many code, library regions, and datacenter workloads like databases as 2 MB huge pages for TLB efficiency, but a single write-heavy object anywhere in a huge page makes the whole page ineligible for migration to LtRAM. Once a page has moved to LtRAM, treating the huge page as the migration unit preserves the TLB benefit, but a single write anywhere in it triggers a 2 MB CoW migration back to DRAM, inflating exactly the write traffic the interface is trying to avoid. Splitting a candidate huge page into 4 KB sub-pages restores fine-grained CoW and lets the read-mostly sub-pages live in LtRAM, at the cost of extra page-table entries and TLB pressure. Transparent huge pages [5] further complicate matters, as `khugepaged` could potentially combine read-mostly pages with one write-heavy page, forcing the whole huge page onto DRAM. Prior software-only tiering work offers potential solutions, such as splitting huge pages with sufficiently skewed access patterns [16] and using a runtime system to dynamically co-locate hot objects on the same pages [2]. When to split, when to coalesce, and which workloads can tolerate the lost huge-page coverage are left for future work.

The interface proposed here is abstracted away from the end user, requiring no application-level changes. An application may hint that a region is read-only or read-mostly (e.g., via `madvise`) so the kernel places it on LtRAM immediately rather than inferring its read-mostly status over time, but such hints are advisory—an application need not change to use AROM. However, a richer hint interface could allow applications to get better performance by enriching the LtRAM logic with domain-specific information, as in existing tiering work like AIFM [26]. For instance, database engines are incredibly well-tuned and optimized with much information about data access patterns. Handing this information off to the kernel for use in LtRAM placement decisions would improve data categorization and reduce the number of migrations back to DRAM. Although automatic management by the operating system is the default, there is a wide landscape of potential optimizations for intelligent applications which can explicitly influence over data placement.

8 Conclusion

DRAM dominates server cost while its cost-per-bit scaling has slowed. LtRAM is one solution, but DRAM-compatible interfaces incur too much overhead. We propose a thin HW/SW interface that safely shifts controller responsibilities into the OS by enforcing an invariant we call Application Read-Only Memory (AROM): applications observe LtRAM as read-only memory, and only the kernel writes to LtRAM during page migrations. The controller exposes cache-line reads, 4 KB page writes, and cell-health metadata; the OS handles placement, migration, and wear-leveling. We prototype this interface on the Enzian platform using NOR flash, placing LtRAM alongside DRAM as main memory for read-mostly data to lower DRAM footprint and per-server cost. Projecting these measurements onto a DDR-class interconnect, LtRAM technologies read within a small multiple of DRAM's latency, from 3.2× for 3D FeFET down to 9% faster than DRAM for STT-MRAM, a 26–79% improvement over Optane.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback on an earlier version of this paper, and the members of the Stanford Differentiated Access Memories Project for many helpful discussions. This work was partially supported by a Samsung Electronics Visiting Scholar Fellowship and the Stanford MemoryDAX Affiliates program.

References

[1] Ying Bai, Huaqiang Wu, Ronggui Wu, Yan Zhang, Ning Deng, Zhiping Yu, and He Qian. 2014. Study of Multi-level Characteristics for 3D Vertical Resistive Switching Memory. *Scientific Reports* 4 (2014), 5780. doi:10.1038/srep05780

[2] Vinay Banakar, Suli Yang, Kan Wu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Kimberly Keeton. 2026. OBASE: Object-Based Address-Space Engineering to Improve Memory Tiering. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. USENIX Association, Seattle, WA, 151–166. https://www.usenix.org/conference/osdi26/presentation/banakar

[3] Yu-Der Chih, Yi-Chun Shih, Chia-Fu Lee, Yen-An Chang, Po-Hao Lee, Hon-Jarn Lin, Yu-Lin Chen, Chieh-Pu Lo, Meng-Chun Shih, Kuei-Hung Shen, Harry Chuang, and Tsung-Yung Jonathan Chang. 2020. A 22nm 32Mb Embedded STT-MRAM with 10ns Read Speed, 1M Cycle Write Endurance, 10 Years Retention at 150°C and High Immunity to Magnetic Field Interference. In *2020 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 222–224. doi:10.1109/ISSCC19947.2020.9062955

[4] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. 2022. Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, 434–451. doi:10.1145/3503222.3507742

[5] Jonathan Corbet. 2009. Transparent hugepages. https://lwn.net/Articles/359158/. Accessed: 2026-08-15.

[6] Counterpoint Research. 2026. Memory Prices Surge Up to 90% From Q4 2025. Counterpoint Research insight report. https://counterpointresearch.com/en/insights/Memory-Prices-Surge-Up-to-90-From-Q4-2025.

[7] K. Florent, M. Pesic, A. Subirats, K. Banerjee, S. Lavizzari, A. Arreghini, L. Di Piazza, G. Potoms, F. Sebaai, S. R. C. McMitchell, M. Popovici, G. Groeseneken, and J. Van Houdt. 2018. Vertical Ferroelectric HfO2 FET based on 3-D NAND Architecture: Towards Dense Low-Power Memory. In *2018 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2.5.1–2.5.4. doi:10.1109/IEDM.2018.8614710

[8] Daewon Ha, Wonsok Lee, M.H. Cho, M. Terai, S.-W. Yoo, H. Kim, Y. Lee, S. Uhm, M. Ryu, C. Sung, Y. Song, K. Lee, S.W. Park, K.-S. Lee, Y.S. Tak, E. Hwang, J. Chae, C. Im, S. Byeon, M. Hong, K. Sim, W.J. Jung, H. Ryu, M.J. Hong, S. Park, J. Park, Y. Choi, S. Lee, G. Woo, J. Lee, D.S. Kim, B.J. Kuh, Yu Gyun Shin, and Jaihyuk Song. 2023. Highly Manufacturable, Cost-Effective, and Monolithically Stackable 4F2 Single-Gated IGZO Vertical Channel Transistor (VCT) for sub-10nm DRAM. In *2023 International Electron Devices Meeting (IEDM)*. 1–4. doi:10.1109/IEDM45741.2023.10413772

[9] J.W. Han, S.H. Park, M.Y. Jeong, K.S. Lee, K.N. Kim, H.J. Kim, J.C. Shin, S.M. Park, S.H. Shin, S.W. Park, K.S. Lee, J.H. Lee, S.H. Kim, B.C Kim, M.H. Jung, I.Y. Yoon, H. Kim, S.U. Jang, K.J. Park, Y.K. Kim, I.G. Kim, J.H Oh, S.Y. Han, B.S. Kim, B.J. Kuh, and J.M. Park. 2023. Ongoing Evolution of DRAM Scaling via Third Dimension -Vertically Stacked DRAM -. In *2023 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*. 1–2. doi:10.23919/VLSITechnologyandCir57934.2023.10185290

[10] Takahiro Hirofuchi and Ryousei Takano. 2020. A Prompt Report on the Performance of Intel Optane DC Persistent Memory Module. *CoRR* abs/2002.06018 (2020). https://arxiv.org/abs/2002.06018

[11] Sumio Ikegawa, Frederick B. Mancoff, Jason Janesky, and Sanjeev Aggarwal. 2020. Magnetoresistive Random Access Memory: Present and Future. *IEEE Transactions on Electron Devices* 67, 4 (2020), 1407–1419. doi:10.1109/TED.2020.2965403

[12] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). http://arxiv.org/abs/1903.05714

[13] Luke James. 2026. Memory will consume 30% of hyperscaler AI data center spending this year, a 4X increase over 2023 - Nvidia gets preferential supply terms well below standard market rates, says analyst firm. https://www.tomshardware.com/tech-industry/memory-will-consume-30-percent-of-hyperscaler-spending-this-year. Accessed: 2026-08-19.

[14] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. 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 (ASPLOS '19)*. Association for Computing Machinery, 317–330. doi:10.1145/3297858.3304053

[15] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. Association for Computing Machinery, 2–13. doi:10.1145/1555754.1555758

[16] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, 17–34. doi:10.1145/3600006.3613167

[17] Sergey Legtchenko, Ioan Stefanovici, Richard Black, Antony Rowstron, Junyi Liu, Paolo Costa, Burcu Canakci, Dushyanth Narayanan, and Xingbo Wu. 2025. Storage Class Memory is Dead, All Hail Managed-Retention Memory: Rethinking Memory for the AI Era. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems (HotOS '25)*. Association for Computing Machinery, 111–118. doi:10.1145/3713082.3730381

[18] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery, 574–587. doi:10.1145/3575693.3578835

[19] Peijing Li, Muhammad Shahir Abdurrahman, Rachel Cleaveland, Sergey Legtchenko, Philip Levis, Ioan Stefanovici, Thierry Tambe, David Tennenhouse, Caroline Trippel, and H.-S. Philip Wong. 2025. Towards Memory Specialization: A Case for Long-Term and Short-Term RAM. In *Proceedings of the 3rd Workshop on Disruptive Memory Systems (DIMES '25)*. Association for Computing Machinery, New York, NY, USA, 27–36. doi:10.1145/3764862.3768175

[20] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. 2025. Tiered Memory Management Beyond Hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 731–747. https://www.usenix.org/conference/osdi25/presentation/liu

[21] Sihang Liu, Suraaj Kanniwadi, Martin Schwarzl, Andreas Kogler, Daniel Gruss, and Samira Khan. 2023. Side-Channel Attacks on Optane Persistent Memory. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 6807–6824. https://www.usenix.org/conference/usenixsecurity23/presentation/liu-sihang

[22] Anni Lu, Junmo Lee, Tae-Hyeon Kim, Muhammed Ahosan Ul Karim, Rebecca Sejung Park, Harsono Simka, and Shimeng Yu. 2024. High-speed emerging memories for AI hardware accelerators. *Nature Reviews Electrical Engineering* 1, 1 (2024), 24–34. doi:10.1038/s44287-023-00002-9

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

[24] Micron Technology. 2022. MT35XU02G 2Gb, 1.8V Serial NOR Flash Memory Datasheet. Datasheet. Part number MT35XU02GCBA.

[25] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, 392–407. doi:10.1145/3477132.3483550

[26] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[27] TrendForce. 2026. Memory Price Outlook for 1Q26 Sharply Upgraded; QoQ Increases of All Product Categories to Hit Record Highs. TrendForce Corp. press release. https://www.trendforce.com/presscenter/news/20260202-12911.html.

[28] Midhul Vuppalapati and Rachit Agarwal. 2024. Tiered Memory Management: Access Latency is the Key!. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. Association for Computing Machinery, 79–94. doi:10.1145/3694715.3695968

[29] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 496–508. doi:10.1109/MICRO50266.2020.00049

[30] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the Performance of Intel Optane Persistent Memory: A Close Look at Its On-DIMM Buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery, 488–505. doi:10.1145/3492321.3519556

[31] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the Challenges of Crossbar Resistive Memory Architectures. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 476–488. doi:10.1109/HPCA.2015.7056056

[32] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[33] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 37–56. https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong