Zswap Boost: Accelerating Transparent Memory Compression in the Cloud

Zheng Liu
Zhejiang University
Hangzhou, China
Alibaba Group
Hangzhou, China
wenqing.lz@alibaba-inc.com

Teng Ma✉
Alibaba Group
Hangzhou, China
sima.mt@alibaba-inc.com

Chengkun Wei✉
Zhejiang University
Hangzhou, China
weichengkun@zju.edu.cn

Ning Zhang
Alibaba Group
Hangzhou, China
chris.zn@alibaba-inc.com

Xingyu Liu
Zhejiang University
Hangzhou, China
lxyu@zju.edu.cn

Gang Deng
Alibaba Group
Hangzhou, China
denggang.dg@alibaba-inc.com

Shangming Cai
Alibaba Group
Hangzhou, China
caishangming.csm@alibaba-inc.com

Wenzhi Chen
Zhejiang University
Hangzhou, China
chenwz@zju.edu.cn

Abstract

Memory is one of the most expensive and underutilized resources in cloud datacenters. Transparent memory compression reclaims overcommitted memory without additional hardware or application changes. Yet our production deployment of Linux zswap reveals substantial overhead for latency-sensitive workloads. Although compressed pages remain in memory, zswap retains disk-oriented swap indexing, red-black-tree lookup, swap-cache management, and slot allocation. Consequently, roughly 40% of swap-in critical-path CPU cycles are spent on bookkeeping rather than decompression.

To eliminate this disk-centric tax, we present Zswap Boost, which encodes the compressed-page metadata location in the page-table entry. It reduces zswap's three-level lookup to two and removes swap-slot, red-black-tree, and swap-cache operations from the critical path. We also integrate Intel IAA using batched compression and asynchronous decompression. Zswap Boost is implemented in our commercial Linux distribution and has run in production for nearly two years. In controlled experiments, Zswap Boost reduces swap-in latency from 6.6 ∼ 19.9 μs to 2.5 ∼ 4.3 μs, cuts Flink kernel-mode CPU time from 54.3% to 19.0%, and saves 12.5% of memory at a 2.13% performance cost. It also reduces Redis P99 latency by up to 18.5% compared with vanilla zswap. In our Flink cluster, the broader memory-overcommit pipeline enabled 130 physical machines to be retired.



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857053

CCS Concepts: • **Software and its engineering** → *Operating systems*; **Memory management**; • **Computer systems organization** → **Cloud computing**.

Keywords: transparent memory compression, zswap, memory swapping, page table, operating systems, cloud computing, memory management

ACM Reference Format:
Zheng Liu, Teng Ma, Chengkun Wei, Ning Zhang, Xingyu Liu, Gang Deng, Shangming Cai, and Wenzhi Chen. 2026. Zswap Boost: Accelerating Transparent Memory Compression in the Cloud. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15-18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3828523.3857053

1 Introduction

In the cloud, the memory budget of a workload is typically provisioned for its peak load. A workload whose working set exceeds its provisioned memory becomes I/O-bound through memory swapping. To meet low-latency requirements while avoiding the out-of-memory (OOM) killer, cloud service providers (CSPs) take a cautious approach. Tenants overprovision virtual-machine (VM) memory to keep systems running without touching swap, and operators often disable swap space entirely to avoid I/O-induced jitter. This is safe but wasteful, and it leaves a large fraction of costly datacenter memory stranded and idle, especially in public clouds.

A common remedy is to enlarge the effective memory of a machine (a.k.a. memory extension). Tiered memory [7, 19, 21] integrates slower local devices such as NVMe or persistent memory (e.g., Intel Optane DC [9]), but requires additional hardware and thus extra cost. Remote memory instead borrows idle memory across machines via interconnects such

Table 1. Comparison of memory-extension mechanisms.

Memory-Extension	Extra-Hardware	Performance	Disk-Centric	Cloud-Deployed
Swap	✘	Low	✔	Hard
Tiered Memory	✔	High	✘	Hard
Remote Memory	✔	Mid	✘	Hard
Zswap	✘	Low	✔	Easy
Zswap Boost	✘	High	✘	Easy

as remote direct memory access (RDMA) or rack-scale Compute Express Link (CXL) [1, 4, 8, 15, 22, 29]. Even with cloud-grade 400 Gbps ConnectX-6 RDMA network interface cards (RNICs), remote accesses are far slower than local DRAM. Across our clusters, we observed 20%∼50% slowdowns at a 1:1 local-to-remote ratio. Remote memory is also hard to operate in the cloud, as cross-machine memory management adds error handling and synchronization complexity. Table 1 summarizes these trade-offs.

Transparent memory compression is an attractive alternative because it is purely software and needs no extra hardware [2, 11]. Zswap [28], a mainline Linux feature, intercepts pages on the swap-out path through the frontswap API [18], compresses them, and stores them in a dynamically sized in-memory pool (zpool) rather than writing them to disk. On a subsequent page fault, it decompresses the payload into a newly allocated page and then installs that page into the faulting process's page table. By keeping reclaimed pages in DRAM, zswap smooths the I/O storms and performance cliffs that plague disk-backed swap, and it has been shown effective at warehouse scale [13, 29]. Recent work also uses hotness-aware, size-adaptive compression to reduce compressed-swap latency and CPU use on mobile devices [16]. Zswap Boost instead targets the Linux swap-path bookkeeping that affects cloud workloads. In practice a userspace agent monitors metrics such as latency and tunes how much memory the system reclaims, bounding performance degradation.

We deployed zswap across our production fleet of thousands of servers (§2.3) and confirmed its memory benefits, but also discovered persistent overhead. Zswap executes substantial swap bookkeeping on every swap-in (§2.4), and this overhead (§2.5) becomes a first-order bottleneck for memory-intensive workloads such as Flink and latency-sensitive workloads such as Redis. Through in-depth on-production profiling, we find that roughly 40% of the CPU cycles on the swap-in critical path are spent not on (de)compression but on swap machinery itself.

The root cause is architectural. Unlike prior work that attacks the symptom by classifying hot/cold pages to limit how much is compressed, we revisit the zswap implementation in vanilla Linux and find that, although compressed pages never leave DRAM, zswap is still built on Linux's disk-centric swap subsystem (§2.1). It reuses `swap_info_struct`, the `type+offset` indexing, a per-type red-black tree, and the swap-slot allocator and swap cache, all of which were designed for block devices. Locating a compressed page therefore requires lookups by `type`, `offset`, and `handle`, together with a tree lock, even though the processor could address the page directly. This indirection is the *disk-centric tax* that inflates swap-in latency and, in turn, the tail latency governed by service-level agreements (SLAs).

Motivated by this finding, we present Zswap Boost, a lightweight swap system that decouples transparent memory compression from disk-oriented swap. Its key insight is simple but consequential. Because a compressed page is resident and directly addressable in memory, its location can be encoded in the page table entry (PTE) itself. Zswap Boost makes two changes (§3.1). First, it repurposes the otherwise-unused `offset` bits of the swap PTE to store a direct pointer to the compressed-page metadata, collapsing the three-level lookup into two levels while removing the red-black tree and its lock from the critical path. Second, it streamlines the swap-out and swap-cache paths by skipping swap-slot scanning and swap-cache indexing that serve no purpose for in-memory data. Crucially, these changes preserve the existing PTE format and swap semantics, so Zswap Boost remains fully transparent to applications and coexists with vanilla swap.

We implement Zswap Boost in the Linux kernel (§3.4) by introducing new indexing data structures (§3.5) and optimizing the swapping workflow (§3.2). The implementation touches 21 files (36 including the IAA extension) and is transparent to applications. We also integrate the Intel In-Memory Analytics Accelerator (IAA) as a complementary extension, using batching for compression and overlapping asynchronous decompression with swap-in metadata processing (§3.6). Zswap Boost has been integrated into our commercial Linux distribution and has run in production for nearly two years. We evaluate it (§4) using Nexmark [20] on a Flink cluster [3] as a platform-as-a-service (PaaS) workload, and redis-benchmark [24] on Redis [23] as an infrastructure-as-a-service (IaaS) workload. Zswap Boost reduces Flink swap-in latency from 6.6 ∼ 19.9 μs to 2.5 ∼ 4.3 μs and Redis get-path latency from 3.51 μs to 1.44 μs (256 B values), cuts total Flink execution time by 195.8 s (7.39%) over zswap, and lowers Redis P99 latency by up to 18.5%. Under the evaluated 25% reclaim configuration, it provides 12.5% capacity-equivalent memory savings.

Our contributions are as follows:

(1) Through in-depth profiling of vanilla zswap on a production Flink fleet, we identify and quantify the disk-centric tax. A substantial fraction of swap-in CPU cycles is spent on disk-oriented swap bookkeeping rather than on decompression, even though compressed pages stay in DRAM.

(2) We design Zswap Boost. To the best of our knowledge, it is the first system to eliminate zswap's disk-centric design and dependencies by encoding the compressed-page location

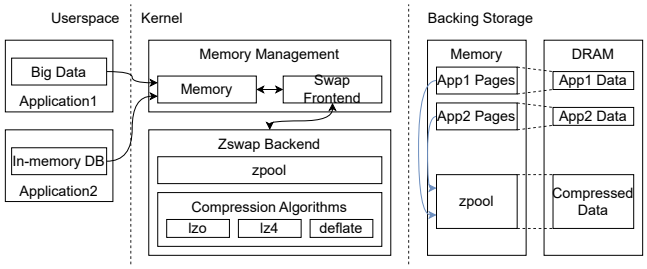


Figure 1. Zswap Architecture

directly in the PTE. We also explain how the design preserves swap correctness and transparency.

(3) We implement Zswap Boost in the Linux kernel and evaluate it with microbenchmarks and production applications (Flink and Redis), reporting both controlled experiments and nearly two years of production deployment experience.

(4) We integrate Intel IAA hardware compression into Zswap Boost (§3.6, §4.4) and show that software streamlining and hardware offloading are complementary. IAA further reduces swap-in latency to 1.8 μs (91% below vanilla), increases compression throughput by 5.9×, and raises memory savings from 12.5% to 15.2%.

2 Background and Motivation

In this section, we first introduce zswap and the disk-centric swap mechanism it is built on in vanilla Linux. We then report the effect of enabling zswap on our production Flink fleet and explain why it triggers more page faults and swapping. Finally, our profiling shows that a large share of swap-in time is spent in kernel swap bookkeeping rather than in decompression.

2.1 Introduction to Zswap

Zswap is a Linux memory-management feature that provides transparent, page-level compression. Its goal is to compress cold pages instead of paging them to disk, improving effective memory capacity under pressure while reducing reliance on slow swap I/O.

When a system, or a cgroup-limited application [27], exhausts its memory budget, the Linux kernel reclaims inactive pages by swapping them to disk-backed swap space, freeing the corresponding page frames (a.k.a. memory swapping). Because disk is orders of magnitude slower than DRAM, frequent swapping severely degrades performance. For this reason, swapping is disabled by default on our production systems.

Figure 1 shows the zswap architecture. Rather than writing a page that is about to be swapped out directly to disk, zswap compresses it and stores it in a dedicated in-memory pool (zpool [28]). When the page is accessed again, the kernel decompresses it and remaps it into the page table, avoiding

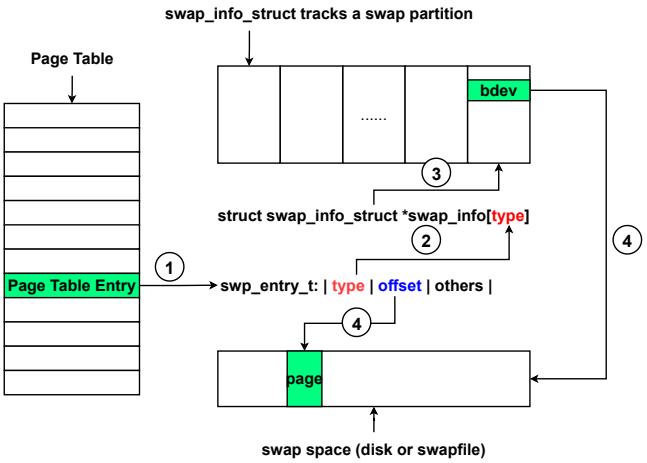


Figure 2. Data Structures of Swap in Vanilla Linux

the latency spikes of disk I/O. This effectively enlarges available memory at the cost of CPU cycles for (de)compression. Zswap thus strikes a balance between paging fully to disk and keeping everything resident, which makes it well suited to memory-intensive and latency-sensitive workloads such as big-data engines and in-memory databases.

2.2 Disk-Centric Swap in Linux

Swap mechanism. Linux's swap subsystem is fundamentally oriented towards block storage, because the original backing store of swap can only be a disk device or a swapfile.

Figure 2 shows the core data structures. The kernel tracks each active swap area with a `swap_info_struct` and indexes a swapped-out page with a `swp_entry_t`, an unsigned long (typically 64 bits) that can be stored directly in a page table entry (PTE). A `swp_entry_t` identifies a page by its swap area (`type`) and position within that area (`offset`). On a fault, the kernel ① reads the PTE and converts it to a `swp_entry_t`. It then ②③ uses `type` to locate the `swap_info_struct` and ④ uses `offset` to read the page. On swap-out, the kernel first scans `swap_info_struct` for a free slot. It then encodes `type` and `offset` into the swap PTE, writes the page, and releases the frame.

Design of zswap. To integrate transparently with this subsystem, zswap reuses the same index format and addresses a `zswap_entry` by `type` and `offset`. As Figure 3 shows, a per-area `zswap_tree` selected by `type` holds a red-black tree that indexes `zswap_entry` objects by `offset`. Each `zswap_entry` points into a `zpool`, and a `handle` locates the compressed bytes. Accessing one compressed page requires three lookup levels: `type`, `offset`, and `handle`. The kernel ① reads `swp_entry_t` from the PTE, ② locates `zswap_tree` using `type`, ③④ finds `zswap_entry` using `offset`, and ⑤ decompresses through `zpool` and `handle` into a newly allocated frame.

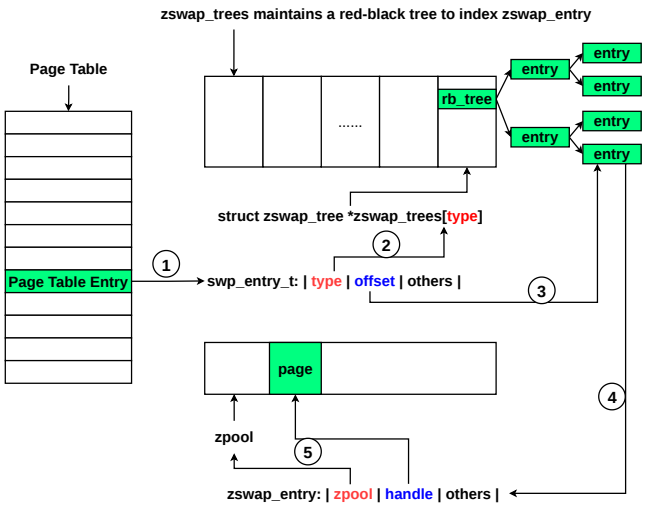


Figure 3. Data Structures of Zswap in Vanilla Linux

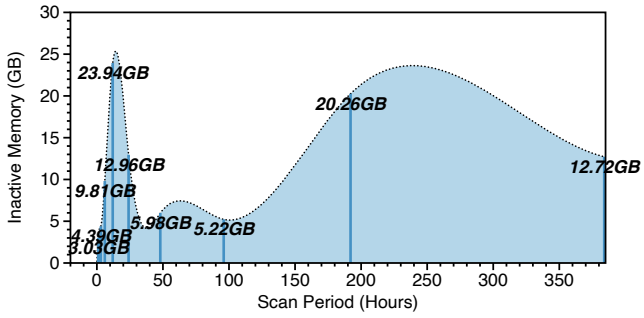


Figure 4. Inactive Memory in Flink

This indexing is necessary for disk-backed swap, where pages leave memory and the processor cannot address them directly. But compressed pages managed by zswap remain in DRAM and are directly addressable. The `type+offset` indirection, the per-area tree, and its lock are thus pure overhead inherited from the disk-centric design. This observation motivates the design in §3.

2.3 Zswap in Production Deployment

Before enabling zswap, we characterized memory usage across our Flink cluster of roughly one thousand servers and analyzed a representative node. Figure 4 reports inactive memory over time on a 512 GB Flink server. More than 25 GB remains untouched for over 24 hours. Because zswap stores reclaimed pages in the zpool rather than on disk, and such cold pages are rarely re-accessed, we expected that careful application selection would allow zswap to avoid paging to disk without violating latency SLAs.

Guided by this analysis, we enabled zswap on the Flink cluster. Flink [26] is a memory-hungry big-data engine whose CPU utilization is effectively capped by memory capacity.

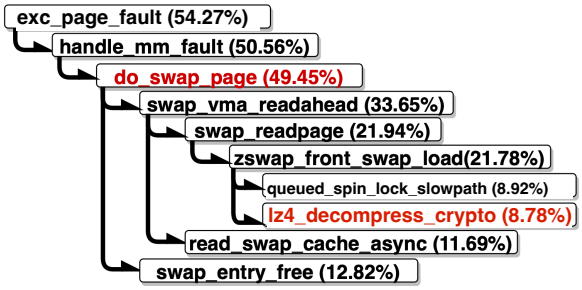


Figure 5. Inclusive call-path sample shares for Flink swap-in with vanilla zswap on Linux 5.10.

Reclaiming idle memory lets us pack more jobs per server, which raises CPU utilization and overall cluster efficiency. In production, this lifted CPU utilization by about 15%. This 15% increase is the aggregate result of our memory-overcommit pipeline, which combines cold-page selection, direct zero-page reclamation, and compressed-memory reclamation. In production, we use the reclaimed capacity to consolidate more jobs while keeping node memory utilization below a 75% safety watermark and meeting workload SLAs.

2.4 Issues in Zswap

Enabling zswap also introduced an unexpected cost. Compared with a node without zswap, a zswap-enabled node exhibits markedly more page faults and swapping. Because the memory available to applications shrinks, accesses more often hit pages already moved into zswap, and each access is handled through the page-fault path. We anticipated some overhead from page faults and (de)compression, but the measured slowdown exceeded our expectations.

Zswap handling splits into two halves. Swap-out (1) scans and selects pages to reclaim, then (2) compresses anonymous pages into the zpool. Swap-in reverses these steps. It (1) locates and decompresses the page in the zpool using the PTE and (2) installs it into the faulting process's page table. We hypothesized that the dominant cost lies in swap-in because reclamation runs in the background. The `kswapd` thread scans and swaps out pages once free memory falls below a watermark [5]. Swap-out is therefore largely off the application's critical path, whereas swap-in is synchronous and latency-critical.

2.5 Profiling of Memory Swap-in

To pinpoint the swap-in bottleneck, we built a test environment matching our online Flink configuration (detailed in §4.1) and profiled it under Nexmark [20] workloads. This environment uses Linux 5.10 because it was the production kernel in our internal systems when the profiling was conducted.

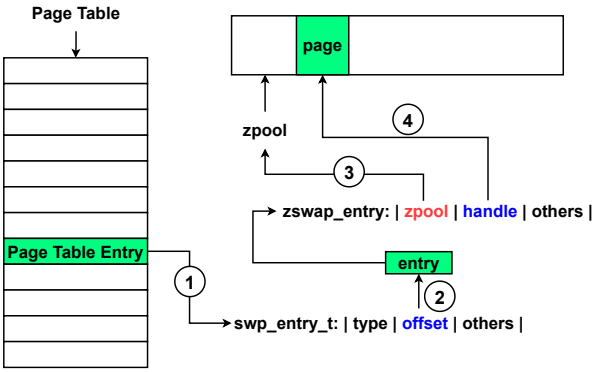


Figure 6. Design of Zswap Boost. The per-area `zswap_tree` and red-black tree are removed. The swap PTE points directly to a `zswap_entry`.

Figure 5 shows function-level samples of the swap-in path captured with `perf-record(1)` [17]. The profile reports inclusive call-path sample shares (the `Children` column of `perf report`). Page-fault handling appears in 54.27% of samples, the swap-in handler in 49.45%, and LZ4 decompression in 8.78%. Because these paths are nested, the percentages are not additive. The 40.67 percentage-point gap between the swap-in and decompression shares indicates that most sampled swap-in work lies elsewhere in the swap path, including page management, kernel data-structure operations, and slot/cache management.

This is the crux of zswap's limitation. Although compressed pages reside in memory, zswap still manages them with `swap_info_struct` and `type`/`offset` indexing designed for block devices. This design requires lookups by `type`, `offset`, and `handle`, together with a tree lock, instead of direct addressing. We call this overhead the disk-centric tax. Zswap Boost aims to eliminate this tax and reduce swap-in latency.

3 Zswap Boost Design

This section first presents the high-level design, workflow optimizations, and correctness properties of Zswap Boost. It then describes the Linux implementation, PTE encoding, and IAA extension.

3.1 Design Overview

The profiling in §2.5 shows that zswap's overhead comes from the disk-centric indirection it inherits, not from compression. This result motivates our central idea. Because a compressed page resides in DRAM and is directly addressable, we store its location in the swap PTE and access its metadata without walking a per-area red-black tree.

Concretely, Zswap Boost stores a direct pointer to a `zswap_entry` in the `offset` field of `swp_entry_t`. In Linux 5.10, vanilla zswap uses the red-black tree to map a swap offset to the corresponding `zswap_entry`, because the offset identifies a backing slot rather than the metadata address. By storing the metadata address in the PTE instead, Zswap Boost makes this offset-to-entry index unnecessary. This yields two benefits. First, the kernel no longer looks up the red-black tree in `zswap_tree` to find a `zswap_entry`, eliminating that entire code path. Second, it no longer takes the lock protecting that tree, removing a source of lock contention on the swap-in critical path. Once these are gone, all disk-related swap structures become dead weight for compressed pages and can be removed, decoupling transparent memory compression from the swap partition.

Figure 6 shows the resulting design. The red-black tree and its associated structures are removed, reducing three lookup levels to two. When an application touches a compressed page, the kernel ① reads `swp_entry_t` from the PTE, ② obtains `zswap_entry` from `offset`, ③ locates the zpool, and ④ decompresses and installs the page.

Removing `zswap_tree` raises one question: how does the kernel recover a `zswap_entry` from a PTE without it? Zswap Boost stores the address of the `zswap_entry`, rather than the object itself, in the PTE. Specifically, it places the low 48 bits of the address in the 50-bit logical `offset` field of `swp_entry_t`. On a fault, the kernel reads the PTE and dereferences this pointer directly. Section 3.5 details how this address is packed into `swp_entry_t` without altering the PTE format.

3.2 Workflow Design

Zswap Boost removes unnecessary work from memory swapping and the swap cache.

Memory swapping. As described in §2.2, swap-out normally scans `swap_info_struct` for a free slot before writing a page. Since Zswap Boost never touches a swap area, it safely skips this step, removing the bitmap scan and slot-allocation overhead entirely. Consequently, Boost-managed pages are not written back to a swap device. Like a zram configuration without backing-device writeback, Zswap Boost relies on bounded reclamation and reserved memory headroom rather than disk-backed OOM resilience. To distinguish Boost entries on swap-in, we reserve the last swap type (`MAX_SWAPFILES-1`) as a marker in the PTE. Reading `swp_entry_t` immediately reveals whether Zswap Boost manages the page. The kernel can then skip the `swap_info_struct` lookup and related bookkeeping.

Swap cache. The swap cache tracks pages that have been read back from swap but not yet evicted, so repeated accesses hit DRAM instead of re-reading swap. It also indexes pages by `offset` in a radix-tree/XArray [6], adding overhead that is pointless for in-memory data. Zswap Boost stores a direct pointer to the resident page inside `zswap_entry`. The kernel follows this pointer and bypasses the swap-cache lookup altogether.

3.3 Correctness and Transparency

Zswap Boost uses the existing non-present swap-PTE format instead of introducing a new format. This preserves the invariants used by the rest of the kernel. The entry remains a non-present swap PTE. Page-table walks, translation lookaside buffer (TLB) handling, and the `PAGE_PRESENT/PAGE_PROTNONE` fast paths are unaffected. The prototype uses `MAX_SWAPFILES-1` as the Boost marker and does not allocate a backing slot for Boost entries. Our evaluated configuration uses one real swap area and therefore does not conflict with this marker. A general deployment must reserve the marker type or reject a conflicting `swapon` operation. Because Boost entries do not belong to a backing swap area, their mappings must be drained before disabling swap. The pointer stored in `offset` references a reference-counted, slab-allocated `zswap_entry`. Each installed Boost PTE, including a copy created by `fork`, contributes to the entry's map count through `swap_duplicate()`. On swap-in, the fault handler acquires the page-table lock, rechecks the PTE with `pte_same()`, and only then dereferences the decoded entry. Removing a PTE drops its mapping reference, and the entry is released only after its mapping and object references reach zero. Zswap Boost is enabled per node for selected workloads rather than serving as a system-wide replacement for vanilla zswap. Each node runs either vanilla zswap or Zswap Boost, allowing deployment and rollback to proceed independently across the cluster.

3.4 Implementation Overview

We implement Zswap Boost in the Linux kernel. Because the design decouples transparent compression from the disk-centric swap path, we add a self-contained module rather than modifying the existing zswap codebase. We leave the original zswap unchanged so that applications already relying on it in production are unaffected. Relative to the Linux 5.10 branch point, the core Zswap Boost implementation changes 21 files, with 1,294 insertions and 71 deletions. Including the IAA extension, the complete prototype changes 36 files, with 4,789 insertions and 796 deletions.

3.5 PTE Encoding

The original `swp_entry_t` encodes `type` and `offset` as follows:

```
|TYPE(59-63)|~OFFSET(9-58)|0|0|X|X|X|X|F|SD|0|
```

On x86-64, bits 59–63 hold `type`, and physical PTE bits 9–58 hold the bitwise complement of the 50-bit logical `offset`. In bits 0–8, `F` denotes userfaultfd write-protect state, and `SD` denotes soft-dirty state. Bits 0, 7, and 8 remain zero. Zswap Boost preserves the non-present swap-PTE format and changes how the available fields identify a `zswap_entry`.

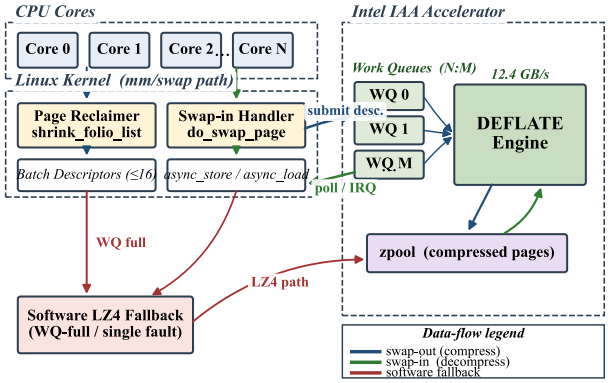


Figure 7. Zswap Boost with Intel IAA. During swap-out, the page reclaimer batches descriptors and maps N CPUs to M IAA work queues (WQs). During swap-in, decompression overlaps with metadata processing. The software LZ4 path handles full WQs and isolated demand faults.

For the 4-level scheme, address bits 0–47 occupy logical offset bits 0–47, while logical offset bits 48–49 are zero. After x86 inversion, physical PTE bits 9–56 contain the complemented address bits and bits 57–58 are one. For the 5-level scheme, the low 50 bits of a 52-bit entry locator occupy the logical offset and the upper two bits occupy software bits 3–4. Decoding recovers the 50-bit offset and concatenates bits 3–4 to reconstruct the locator. The experiments in this paper use the 4-level configuration. The encodings are:

```
swp_entry_t format in 4-level page table
|TYPE(59-63)|1|1|~ENTRY(9-56)|0|0|X|X|X|X|F|SD|0|

swp_entry_t format in 5-level page table
|TYPE(59-63)|~LOC(9-58)|0|0|X|X|E|E|F|SD|0|
```

3.6 IAA Extension

While Zswap Boost eliminates disk-centric bookkeeping, software compression still sits on the reclaim critical path. Recent Intel server CPUs include Intel IAA [10, 12], which offloads DEFLATE through descriptor-driven work queues (WQs). Recent work also integrates Intel QAT with kernel-space memory compression [30]. Such hardware acceleration reduces compression cost, while Zswap Boost targets swap-path bookkeeping. Compared with LZ4, DEFLATE typically yields a higher compression ratio, improving effective memory capacity and reducing zpool footprint. IAA is optimized for throughput rather than single-request latency. Compressing one 4 KB page incurs a fixed descriptor round-trip (~5–6 μs), which can exceed the latency of in-core LZ4 (~2.5 μs). Our design therefore treats IAA as a pipelined device that must be kept busy with batches, while preserving a low-latency software fast path.

We extend Zswap Boost with an IAA backend (Figure 7) that addresses six practical challenges.

N:M CPU-to-WQ mapping and backpressure. We map many CPUs to a smaller number of shared WQs via round-robin selection, avoiding per-core WQ provisioning. If a WQ is saturated, we immediately fall back to software LZ4 to avoid reclaim-path stalls and potential deadlocks.

Asynchronous frontswap store. We add an `async_store` callback to `frontswap_ops`. Swap-out submits compression descriptors without blocking and defers completion checks until batch commit time, allowing reclaim to continue while IAA runs.

Batch compression in reclaim. Inside `shrink_folio_list`, we accumulate up to 16 pages in a per-CPU descriptor ring and submit them as a batch. This amortizes descriptor overhead and improves WQ utilization. For swap-in, the IAA prototype sets `page-cluster=3`, allowing an adaptive local readahead window of up to eight pages to create additional decompression requests.

Latency hiding on swap-in. On a page fault, we first allocate the destination page and then submit the decompression descriptor. While IAA executes, the CPU performs the remaining metadata preparation, including least-recently-used (LRU) insertion and cgroup accounting. The PTE is installed only after decompression completes successfully. If the fault is isolated or the WQ is busy, we use a software LZ4 fallback to bound tail latency.

Non-uniform memory access (NUMA) and work conservation. We prefer a local IAA device for pages reclaimed from a given NUMA node and use a remote device only when local queues are saturated. This minimizes cross-socket traffic while preserving accelerator throughput under pressure.

Memory management and correctness. For compression, we pre-allocate output buffers based on a conservative worst-case bound for DEFLATE output size. On completion, we record the exact compressed length and populate the corresponding `zswap_entry`. For decompression, we target a newly allocated page and expose it to the page table only after successful completion. Any IAA error triggers a software retry. If that retry fails, the existing swap-fault handling propagates the error.

4 Evaluation

Sections 4.2 and 4.3 evaluate software-only Zswap Boost against disabled compression and vanilla zswap under the common setup in §4.1. We examine swap-in latency and kernel-path overhead at the micro level, followed by end-to-end performance for Flink and Redis. Section 4.4 separately evaluates the complementary IAA extension, and §4.5 discusses threats to validity.

4.1 Experimental Setup

We run all controlled experiments on a virtual machine in our public cloud. The VM matches the per-node hardware and software of our production Flink and Redis clusters, while avoiding cluster-level scheduling, network, and replication effects that are orthogonal to the swap path. Zswap Boost is a per-node kernel mechanism, and swap-in is handled locally by the faulting core without cross-node coordination. A single-machine experiment therefore isolates its effect, while our production deployment provides fleet-level context (§4.3, §5).

We evaluate Zswap Boost on a virtual machine with an Intel Xeon Platinum 8163 processor (16 physical cores and 32 logical cores), 128 GB of memory, and a 500 GB NVMe drive. Because the evaluated applications are memory intensive, disk and network performance do not affect the results. The guest VM runs CentOS 8.6 with Linux 5.10.

Benchmarks. We use Flink [26] (v1.15.4) and Redis [23] (v7.2) as representative memory-intensive applications. In our cloud, Flink is a key PaaS service, and Redis is a foundational IaaS service. Redis is compiled with snapshotting disabled to avoid disk I/O. We chose these two because zswap is enabled in both our production Flink and Redis clusters. For Flink, we drive load with `nexmark` [20]. A single-machine Flink deployment is sufficient to isolate this per-node mechanism. For Redis, we use `redis-benchmark` [24] to issue 100 million requests uniformly across all keys.

Configurations. Based on inactive-memory measurements from our production Flink and Redis clusters, we use a 25% anonymous-page reclaim target for both applications while permitting the corresponding memory overcommitment. This target balances memory savings against application performance and matches our production zswap configuration. It leaves enough headroom to prevent accesses to reclaimed data from forcing additional pages to be swapped out.

The compression ratio of the zpool is 2:1, matching the default setting of the kernel. This indicates that the zpool allocates one page to store two compressed pages. The zpool does not need to be reserved at the start of VM initialization, and it can be dynamically expanded or shrunk based on application requirements.

Metrics. We evaluate (1) swap-in latency and kernel-path overhead, and (2) end-to-end performance for memory-sensitive and latency-sensitive applications. To measure swap-in improvements, we instrument the Linux kernel to collect per-function execution times along the swap-in path and use its built-in statistics to analyze latency distributions. In the application benchmarks, `nexmark` runs 22 test cases sequentially and reports total execution time, while `redis-benchmark` reports requests per second (RPS) and latency. The Flink and Redis macrobenchmarks are repeated three times, and their average results are reported.

Baselines. To evaluate the performance loss introduced after enabling transparent memory compression, we disable zswap, without performing memory compression and decompression, as our baseline for comparing real-world Flink and Redis scenarios. We also compare the current zswap implementation in the Linux kernel to determine the improvement effects of the Zswap Boost approach. For the software-only zswap and Boost configurations, we set `/proc/sys/vm/page-cluster=0`, thereby disabling swap-page readahead. Disk-backed swap readahead amortizes block I/O over neighboring swap entries, whereas in-memory compressed objects do not preserve the corresponding storage-layout locality. Speculatively loading neighboring compressed objects therefore adds decompression and memory pressure without the same sequential-I/O benefit (§5.2). We do not include a separate readahead ablation, so the software-only results apply to this setting. Both zswap and Zswap Boost use the zbud zpool type [28] and the LZ4 compression algorithm. We use zbud for its fragmentation resistance in cloud workloads, although the architecture supports other zpool implementations. Sections 4.2 and 4.3 evaluate the software-only LZ4 configurations. Section 4.4 separately evaluates the complementary IAA offload.

4.2 Micro Benchmark

We first evaluate swap-in latency using the Flink and Redis scenarios. We collect latency statistics from `do_swap_page` while running `nexmark` and `redis-benchmark`. After enabling transparent memory compression, the number of swap-in and swap-out events increases. This increase is expected because compressed payloads occupy part of DRAM, increasing pressure on the remaining uncompressed anonymous pages. Applications wait synchronously for swap-in completion, so these operations directly affect performance.

Average latency. Figure 8 reports swap-in latency for the Flink and Redis tests. Flink performs frequent concurrent memory accesses, so zswap reaches 17.54 μs for q0. Zswap Boost removes swap-related data-structure lookups and lock contention, reducing q0's average latency to 4.22 μs, a 75.94% reduction.

Redis uses a single-threaded core processing path and therefore experiences less lock contention. With 256-byte values, zswap's swap-in latency is 3.43 μs for SET and 3.51 μs for GET. Zswap Boost reduces these values to 1.38 μs and 1.44 μs, corresponding to decreases of 59.77% and 58.97%, respectively.

Tail-latency. We use the Linux kernel's statistical functions to collect swap-in latency histograms for the workloads above. Figure 9 shows that Zswap Boost shifts the distributions to the left, indicating lower swap-in tail latency. In Flink, the peak of zswap occurs in the 4~8μs interval with a recorded count of 47.1 million and the peak of Zswap Boost occurs in the 2~4μs interval with a recorded count of 46.0 million. We can also observe this trend in Redis. Three factors contribute to this shift. First, the kernel indexes the corresponding `zswap_entry` directly through `swp_entry_t` in the page table, avoiding manipulation of `zswap_trees` and lock contention on the red-black tree. Second, the `type` field marks a page as managed by Zswap Boost, allowing the free-slot lookup to be skipped. Third, Zswap Boost skips the swap-cache lookup. Together, these operations account for much of the swap-in execution cost.

Table 2. Total Execution Time of Flink

Method	disabled	zswap	Zswap Boost
Time (s)	2401.4	2648.4	2452.6

Kernel mode. We again use `perf-record(1)` to collect kernel-mode function samples. Figure 10 shows the results. Figure 10 reports inclusive call-path sample shares for selected call paths. The page-fault entry share decreases from 54.3% with zswap to 19.0% with Zswap Boost, while the `do_swap_page` share decreases from 49.5% to 14.3%. The LZ4 decompression share remains about 8.2%. Their reduction shows that Zswap Boost removes substantial bookkeeping from the swap-in path. The reduced kernel overhead frees processor resources for other tasks.

4.3 Macro Benchmark

Memory-sensitive. Nexmark runs 22 Flink test cases sequentially and reports both per-case and aggregate execution times. Table 2 and Figure 11 show these results. Each bar in Figure 11 reports the arithmetic mean of three runs.

With memory compression disabled, the total execution time across all tests is 2401.4 s. It increases to 2648.4 s with zswap and 2452.6 s with Zswap Boost, corresponding to increases of 10.29% and 2.13% relative to the no-compression baseline. Zswap Boost reduces execution time by 7.39% compared with zswap.

In our evaluation environment, we try to reclaim 25% of memory space and use the zbud type by default. Zswap Boost achieves 12.5% memory savings compared to no compression, matching its efficiency while reducing overhead. This 12.5% result represents gross capacity-equivalent headroom under our evaluation configuration. It does not imply a linear 12.5% increase in jobs or throughput, which also depends on each workload's CPU demand, memory footprint, and service-level objective (SLO) constraints.

Across test cases, the execution-time change relative to disabled compression ranges from −4.33% to 145.44% with zswap and from −1.82% to 17.91% with Zswap Boost.

Among individual test cases, q10 shows the largest slowdown with both zswap and Zswap Boost. Flink generates substantial page-cache data during q10, leaving less memory available to applications and causing more frequent

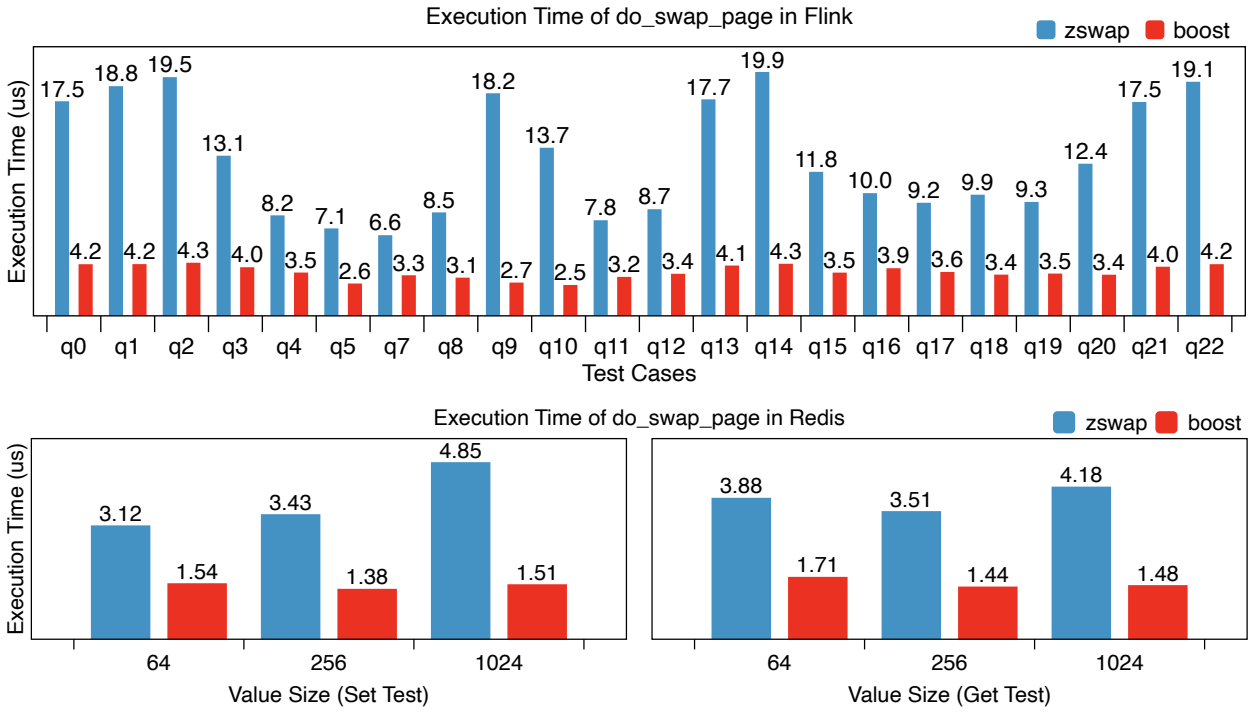


Figure 8. Swap-in latency measured in do_swap_page while running Nexmark and redis-benchmark.

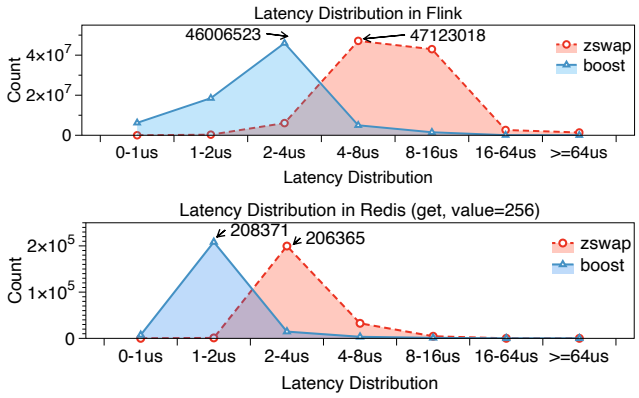


Figure 9. Event-level swap-in latency histograms for Flink and Redis. Each observation represents one swap-in event.

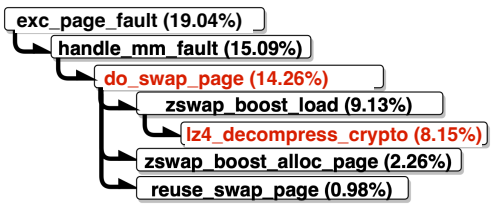


Figure 10. Inclusive call-path sample shares for Flink swap-in with Zswap Boost on Linux 5.10.

anonymous-page reclaim. The resulting swap-in and swap-out operations increase q10's execution time.

Overall, Zswap Boost reduces the performance overhead of memory compression. The magnitude of the improvement varies across the Flink test cases.

Latency-sensitive. We use `redis-benchmark` to evaluate the common SET and GET operations at several value sizes. Throughput decreases as the value size grows because each request accesses more data. Figure 12 reports Redis throughput in RPS. We discuss the 64- and 1024-byte results because the 256-byte results show the same trend. The performance drop is small at 64 bytes but noticeably larger for zswap at 1024 bytes.

At 64 bytes, the disabled configuration achieves 234,956 SET RPS and 252,806 GET RPS. Zswap achieves 227,365 and 243,551 RPS, while Zswap Boost achieves 228,227 and 249,650 RPS. The corresponding improvements over zswap are 0.38% and 2.5%. At 1024 bytes, the disabled configuration achieves 193,798 SET RPS and 211,282 GET RPS. Zswap achieves 160,289 and 192,089 RPS, while Zswap Boost reaches 182,268 and 200,674 RPS. Zswap Boost therefore improves throughput over zswap by 13.7% for SET and 4.5% for GET. Its throughput remains about 5% below the disabled configuration.

Latency is a key SLO metric for Redis. We use `redis-benchmark` to measure the effect of memory compression and the benefit of Zswap Boost. Figure 12 reports average, P50, and P99 latency. Because P99 is most important to our

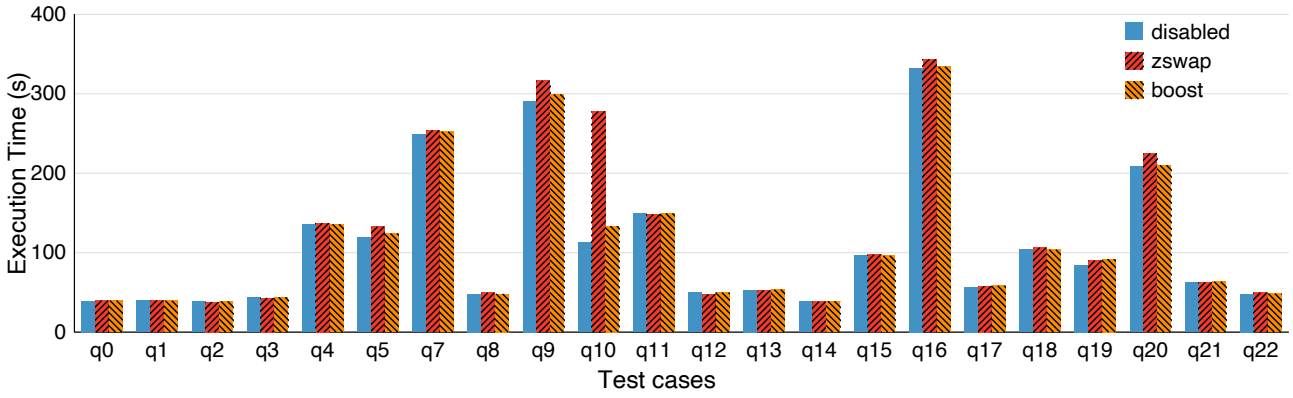


Figure 11. Mean execution time across three runs for each Flink test case.

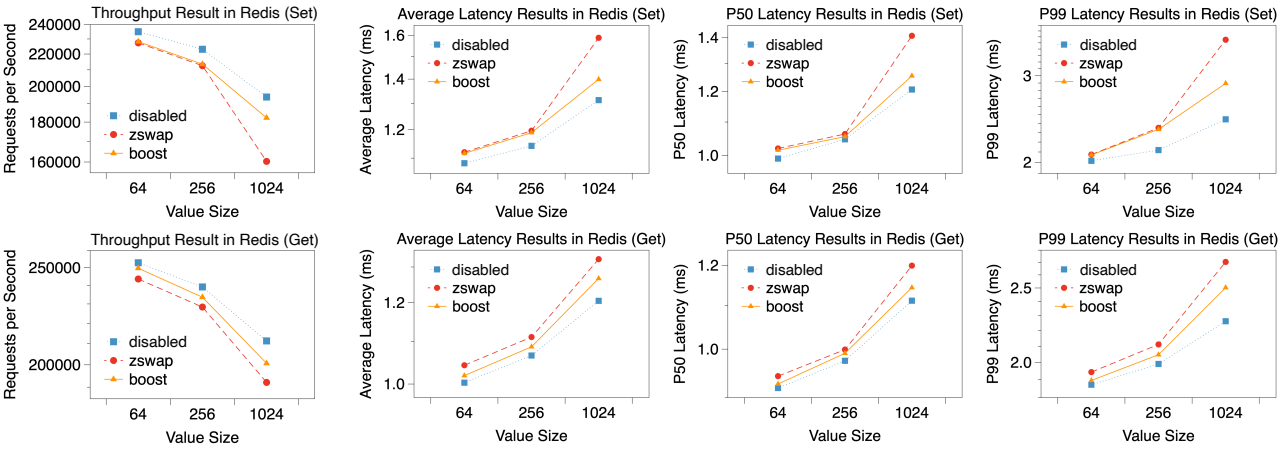


Figure 12. Redis SET/GET throughput (left) and average, P50, and P99 latency (right) across the tested value sizes.

service, we focus on it below. The average and P50 results show the same trends.

Figure 12 shows that the P99 differences between zswap and Zswap Boost are small for 64- and 256-byte values. At 1024 bytes, Zswap Boost reduces P99 latency relative to zswap by 18.47% for SET and 7.40% for GET.

In summary, the evaluations above (i.e., Redis) show that deploying memory compression features can save more memory space in latency-sensitive cloud applications such as key-value stores, databases, and web services.

4.4 IAA Hardware Offloading

We further evaluate the Zswap Boost + IAA prototype (§3.6) on a 4th-gen Intel Xeon Scalable (Sapphire Rapids, 2 IAA devices/socket) using our Linux 5.10 branch. Linux 6.13 is used only for the Intel upstream zswap-IAA comparator. The methodology follows §4.1's 25% reclaim setting, but Boost+IAA stores each compressed payload in a variable-sized buffer rather than a zbud-backed zpool, allowing DEFLATE compression ratios beyond zbud's 2:1 packing limit. This configuration sets `page-cluster=3`, enabling an adaptive local prefetch window of up to eight pages to increase the IAA decompression batch.

Swap-in latency. Figure 13(a) shows the swap-in latency progression. Adding IAA to Zswap Boost further reduces latency from 4.3 μs to 1.8 μs, a 58% reduction over Boost alone. IAA decompression overlaps with memory-cgroup charging, LRU insertion, and reverse-mapping preparation. The destination PTE is installed only after decompression completes.

Compression throughput. Figure 13(b) shows that IAA's DEFLATE achieves 12.4 GB/s, a 5.9× improvement over software LZ4 (2.1 GB/s). During pressure spikes, compression is no longer the reclaim bottleneck.

Compression ratio. On our workloads, DEFLATE averages 2.4:1 versus LZ4's 2.0:1. The corresponding memory saving is measured end-to-end rather than derived from the average ratio. In the Boost+IAA configuration it rises from 12.5% to 15.2% (Table 3).

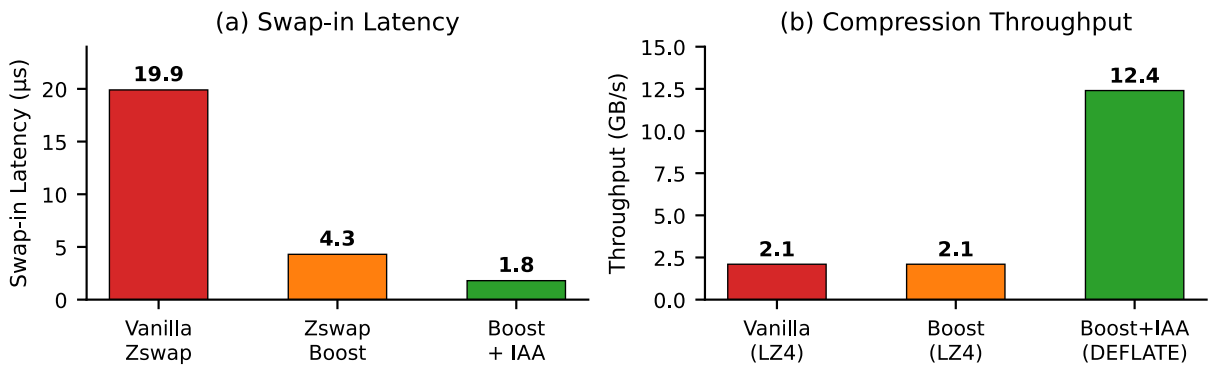


Figure 13. IAA microbenchmark. Panel (a) shows swap-in latency, where IAA reduces latency by a further 58% beyond Boost alone (91% total versus vanilla). Panel (b) shows compression throughput, where IAA achieves 5.9× the software LZ4 rate.

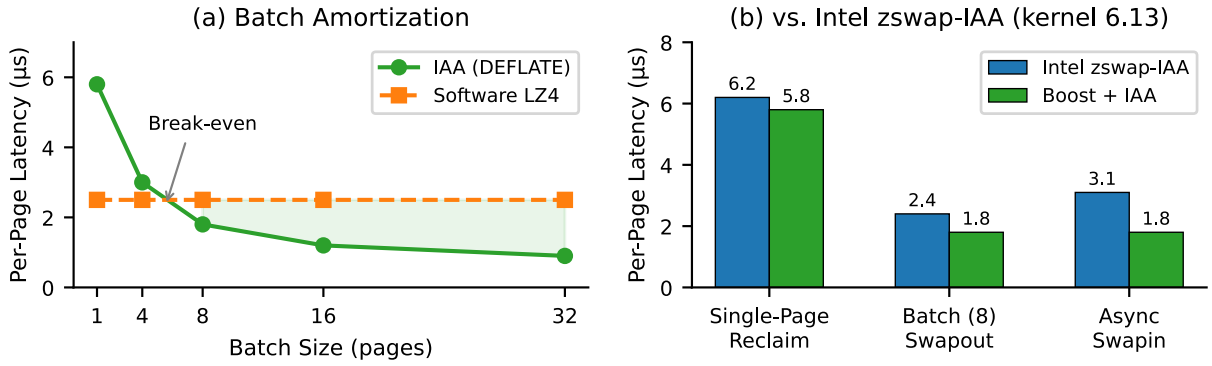


Figure 14. (a) Batch amortization: IAA becomes faster than software LZ4 at batch >4 pages. (b) Zswap Boost + IAA (Linux 5.10) and Intel upstream zswap-IAA (Linux 6.13).

Batch amortization. Figure 14(a) shows per-page latency versus batch size. At batch 1, IAA (5.8 μs) exceeds LZ4 (2.5 μs). At batch >4, IAA becomes faster than LZ4. At batch 16, it reaches 1.2 μs/page. This result informs our hybrid strategy. Software LZ4 handles demand-fault single-page decompression, while IAA handles swap-out batches and prefetch-triggered operations.

Comparison with Intel zswap-IAA. Mainline Linux has included an Intel IAA compression driver through the `crypto_acomp` interface since v6.8 [31]. Sabre uses Intel IAA in host user space to compress and restore serverless microVM snapshots [14]. Recent work also analyzes hardware-accelerated compressed-memory swap across system configurations [25]. Figure 14(b) uses the same hardware but compares Zswap Boost + IAA on Linux 5.10 with Intel's approach on Linux 6.13. It is therefore a cross-kernel comparison. Zswap Boost + IAA is faster in all reported scenarios. The gap is largest in async swap-in (1.8 vs. 3.1 μs, −42%).

Table 3. End-to-end impact of IAA on Flink and Redis. Redis uses 256-byte values.

Metric	Disabled	Zswap	Boost	+IAA
Flink time (s)	2401.4	2648.4	2452.6	2421.8
Overhead	—	+10.3%	+2.1%	+0.8%
Mem. saved	0%	12.5%	12.5%	15.2%
Redis P99 get	1.99 ms	2.11 ms	2.05 ms	1.99 ms
Redis P99 set	2.12 ms	2.35 ms	2.34 ms	2.15 ms

Relative to the Linux 5.10 zswap baseline, Zswap Boost eliminates red-black-tree lookup and bypasses swap-slot allocation and swap-cache indexing. Linux 6.13 replaces the red-black tree with an XArray but retains swap-management overhead. This cross-kernel result is consistent with benefits from reduced swap-management overhead, but does not isolate those benefits from kernel-version and configuration differences. Software streamlining and hardware acceleration are complementary.

Macrobenchmark. Table 3 shows end-to-end results. With IAA, Flink overhead drops to +0.8% (from +2.1% Boost, +10.3% vanilla) while savings rise from 12.5% to 15.2%. Redis GET P99 matches the no-compression baseline, and SET P99 is within 1.4%. The results provide three insights. Hardware offload benefits from software-path streamlining. Batch size is the primary lever for IAA efficiency. DEFLATE's higher ratio provides an additional 2.7 percentage points of savings that compound across fleets.

4.5 Threats to Validity

We highlight three threats and how we mitigate them. **Single-node measurement.** Our controlled experiments use one VM rather than a full cluster. This is intentional. Zswap Boost is a per-node kernel mechanism on the synchronous swap-in path, so cluster-level factors (scheduling, networking, and replication) would add noise without exercising the path we optimize. We mitigate the external-validity gap by configuring the VM identically to production nodes and by corroborating the results with nearly two years of fleet deployment (§5). **Workload coverage.** We focus on Flink and Redis because they are the workloads for which we have production zswap experience. Nexmark's 22 queries and Redis SET/GET operations across value sizes span memory-bound batch and latency-bound serving regimes. The mechanism targets the kernel swap path rather than application logic. **Configuration sensitivity.** Results depend on the reclaim ratio (25%), zpool type (zbud), and compressor (LZ4). We use the same settings as production to keep the comparison representative. Section 5 discusses how `swappiness` and reclaim aggressiveness shift the trade-off.

5 Discussion

We discuss Zswap Boost's deployment experience, limitations, and future work.

5.1 Deployment Experiences

We have also accumulated practical experience in deploying transparent memory compression features on our production systems. Although the page-age, zero-page, and compression-ratio observations below are not a controlled before-and-after comparison between vanilla zswap and Zswap Boost, they characterize the workloads and the overall production cold-page pipeline that informed our deployment choices. Based on data sampling for Flink, we have the following findings.

(1) In all anonymous pages of the Flink application, about 19% of the anonymous pages have not been accessed for more than 12 hours, and about 15% have not been accessed for over 24 hours. Java off-heap memory of Flink is usually accessed less frequently.

(2) Within the anonymous pages of Flink that have not been accessed for over 12 hours, there are a significant number of all-zero anonymous pages, accounting for about 4% of all anonymous pages, while the all-zero anonymous pages not accessed for over 24 hours account for about 3.5%.

(3) The data compression ratio for both Flink and Redis is around 50% using LZ4.

In our largest Flink cluster, cold-page compression reclaimed about 30 TB and lowered the memory watermark by 6.5%. The page-selection policy and resulting capacity benefit are independent of the compression backend, so the same policy applies to vanilla zswap and Zswap Boost. Together with zero-page reclamation, the overall pipeline allowed 130 physical machines (13.75%) to be taken offline while maintaining the 75% safety watermark.

In production, `idlemd` coordinates `kidled` page-age scanning and `coldpgs` reclamation. We replace all-zero pages that have been idle for more than 6 hours with the kernel `ZERO_PAGE`, and compress non-zero anonymous pages that have been idle for more than 24 hours. We consolidate workloads only while node memory utilization remains below the 75% safety watermark, and stop or roll back reclamation if workload SLAs deteriorate.

Operators derive a workload-specific reclaim budget from the measured cold-page volume and configure `idlemd` per cgroup. Because swap-in is synchronous and Zswap Boost has no disk fallback, reclaiming beyond the cold working set can increase tail latency and ultimately cause OOM. We therefore roll out Zswap Boost per node, monitor memory pressure and workload SLOs, and stop reclamation or roll back the node when the configured thresholds are exceeded.

5.2 Limitations

The current tiered memory solutions, including Zswap Boost, still have some limitations. Because swap-in remains synchronous, Zswap Boost cannot eliminate all overhead for latency-sensitive workloads. Under the evaluated 25% reclaim configuration, Redis throughput remains approximately 5% below the no-compression baseline even with Zswap Boost. There are several common issues that tiered memory solutions need to solve.

Our measurements characterize the Linux 5.10 implementation used in our internal systems. Upstream Linux later removed zswap's frontswap attachment in Linux 6.5 and replaced the red-black tree with a per-type XArray and read-copy-update (RCU)-protected lookup in Linux 6.8. These changes remove the tree component but not all swap-slot and swap-cache bookkeeping.

The Linux kernel's transparent memory-compression feature requires a configured swapfile even though compressed pages are not written to it. In future work, we can remove this dependency from Zswap Boost, simplifying activation in production.

The `swappiness` parameter controls the balance between reclaiming anonymous pages and page-cache pages. Because zswap and Zswap Boost operate on anonymous pages, setting `swappiness=0` largely removes their memory-saving benefit. Increasing it can expose more pages to compression, but reclaiming beyond the cold working set increases synchronous swap-ins and may degrade tail latency. Operators should therefore tune `swappiness` together with the workload-specific reclaim budget and idle-age thresholds.

5.3 Future Work

Remote Memory. As a component of tiered-memory systems, zswap can be combined with memory-pooling solutions such as CXL [4]. Currently, zswap stores compressed pages only in local memory. Future CXL or RDMA interconnects could store compressed pages on remote nodes, improving memory utilization in cloud systems.

Prefetching. The current software-only Zswap Boost path has no specialized optimization for loading compressed pages during swap-in. Traditional disk-backed swapping performs readahead to exploit locality among neighboring swap entries and amortize storage I/O. Compressed objects are already in memory and do not preserve the same storage-layout locality, while application access patterns are difficult to predict. Directly adopting disk-oriented readahead can therefore introduce unnecessary decompression and memory pressure, so we disable it in the software-only evaluation. IAA changes this trade-off because its fixed submission cost benefits from additional concurrent requests. The IAA prototype therefore uses `page-cluster=3` to permit an adaptive local prefetch window of up to eight pages. This is a device-amortization mechanism rather than a workload-aware prediction policy. In future work, techniques such as Profile-Guided Optimization (PGO) could identify when neighboring compressed pages are likely to be accessed and provide more selective swap-in prefetching.

Hardware Offloading. We have implemented and evaluated an Intel IAA integration for Zswap Boost (§3.6, §4.4). The results show that software streamlining and hardware acceleration are complementary. By shortening the software path, Zswap Boost allows IAA to handle the remaining compression work more effectively. Open questions remain around per-access-pattern algorithm selection (LZ4 for latency vs. DEFLATE for ratio) and sliding-window prefetch strategies, which we plan to address in follow-up work targeting upstream adoption.

6 Conclusion

Transparent memory compression provides a software-only way to improve memory utilization in the cloud, but Linux zswap retains substantial disk-oriented bookkeeping on its critical path. We find that approximately 40% of sampled swap-in CPU cycles are spent on this bookkeeping. Zswap Boost addresses the overhead by encoding compressed-page locations in PTEs, reducing the lookup path, and removing rbtree, swap-slot, and swap-cache operations. Intel IAA hardware offload further reduces the evaluated overhead to 0.8% while providing 15.2% memory savings. Zswap Boost has run in production for nearly two years. These results show that removing disk-oriented bookkeeping can make transparent memory compression practical for latency-sensitive cloud workloads. Our deployment experience also highlights that workload selection, bounded reclamation, and sufficient memory headroom remain essential for safe operation.

Acknowledgments

We thank the anonymous reviewers, our shepherd, and the artifact evaluators for their valuable feedback that improved this paper. This work was supported by the National Natural Science Foundation of China (Grant No. 62302441), Alibaba Innovative Research Program, Shandong Provincial Higher Education Institutions Youth Innovation Team Development Plan (No. 2025KJH023), and Zhejiang University Education Foundation Qizhen Scholar Foundation.

Artifact Availability

The evaluated artifact (v1.0.0) is archived on Zenodo at https://doi.org/10.5281/zenodo.23221535.

References

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

[2] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proceedings of the linux symposium*. Citeseer, 19–28.

[3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[4] Compute Express Link Consortium, Inc. 2025. HOME | Compute Express Link — computeexpresslink.org. https://www.computeexpresslink.org. [Accessed 2026-10-09].

[5] Jonathan Corbet. 2004. Kswapd and high-order allocations. https://lwn.net/Articles/101230/. [Accessed 2026-10-09].

[6] Jonathan Corbet. 2018. The XArray data structure. https://lwn.net/Articles/745073/. [Accessed 2026-10-09].

[7] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. 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*. 727–741.

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

[9] Intel Corporation. 2022. Intel® Optane™ Persistent Memory 200 Series — intel.com. https://www.intel.com/content/www/us/en/content-details/754277/intel-optane-persistent-memory-200-series-product-brief.html. [Accessed 2026-10-09].

[10] Intel Corporation. 2023. Intel® In-Memory Analytics Accelerator (Intel® IAA) — intel.com. https://www.intel.com/content/www/us/en/content-details/780887/intel-in-memory-analytics-accelerator-intel-iaa.html. [Accessed 2026-10-09].

[11] Seth Jennings. 2013. Transparent memory compression in linux.

[12] Jaeyoung Kang, Qirong Xia, Ipoom Jeong, Yongjoo Park, and Nam Sung Kim. 2025. Intel In-Memory Analytics Accelerator: Performance Characterization and Guidelines. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 124–136. doi:10.1109/ISPASS64960.2025.00021

[13] 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*. 317–330.

[14] Nikita Lazarev, Varun Gohil, James Tsai, Andy Anderson, Bhushan Chitlur, Zhiru Zhang, and Christina Delimitrou. 2024. Sabre: Hardware-Accelerated Snapshot Compression for Serverless MicroVMs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 1–18.

[15] 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*. 574–587.

[16] Yu Liang, Aofeng Shen, Chun Jason Xue, Riwei Pan, Haiyu Mao, Nika Mansouri Ghiasi, Qingcai Jiang, Rakesh Nadig, Lei Li, Rachata Ausavarungnirun, Mohammad Sadrosadati, and Onur Mutlu. 2025. Ariadne: A Hotness-Aware and Size-Adaptive Compressed Swap Technique for Fast Application Relaunch and Reduced CPU Usage on Mobile Devices. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1588–1602. doi:10.1109/HPCA61900.2025.00118

[17] Linux perf wiki Contributors. 2024. Perf Wiki — perf.wiki.kernel.org. https://perf.wiki.kernel.org/index.php/Main_Page. [Accessed 2026-10-09].

[18] Dan Magenheimer. 2020. Frontswap; The Linux Kernel documentation — kernel.org. https://www.kernel.org/doc/html/v5.10/vm/frontswap.html. [Accessed 2026-10-09].

[19] 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*. 742–755.

[20] Nexmark. 2020. GitHub - nexmark/nexmark: Benchmarks for queries over continuous data streams. — github.com. https://github.com/nexmark/nexmark. [Accessed 2026-10-09].

[21] Zhenlin Qi, Shengan Zheng, Ying Huang, Yifeng Hui, Bowen Zhang, Linpeng Huang, and Hong Mei. 2025. Chrono: Meticulous Hotness Measurement and Flexible Page Migration for Memory Tiering. In *Proceedings of the Twentieth European Conference on Computer Systems*. doi:10.1145/3689031.3717462

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

[23] Redis Ltd. 2026. Redis — redis.io. https://redis.io. [Accessed 2026-10-09].

[24] Redis Ltd. 2026. Redis benchmark — redis.io. https://redis.io/docs/management/optimization/benchmarks/. [Accessed 2026-10-09].

[25] Eunyeong Sim and Woongki Baek. 2026. Performance Analysis of Hardware-Accelerated Compressed Memory Swap. In *Euro-Par 2026: Parallel Processing (Lecture Notes in Computer Science, Vol. 16781)*. Springer Nature Switzerland, 388–402. doi:10.1007/978-3-032-35248-4_27

[26] The Apache Software Foundation. 2014. Apache Flink® — Stateful Computations over Data Streams — flink.apache.org. https://flink.apache.org. [Accessed 2026-10-09].

[27] The kernel development community. 2010. Memory Resource Controller; The Linux Kernel documentation — kernel.org. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/memory.html. [Accessed 2026-10-09].

[28] The kernel development community. 2013. zswap; The Linux Kernel documentation — kernel.org. https://www.kernel.org/doc/html/latest/admin-guide/mm/zswap.html. [Accessed 2026-10-09].

[29] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. Tmo: transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 609–621.

[30] Qirong Xia, Houxiang Ji, Yang Zhou, and Nam Sung Kim. 2025. Hardware-Accelerated Kernel-Space Memory Compression Using Intel QAT. *IEEE Computer Architecture Letters* 24, 1 (2025), 57–60. doi:10.1109/LCA.2025.3534831

[31] Tom Zanussi. 2023. crypto: iaa - Add Intel IAA Compression Accelerator crypto driver core. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ea7a5cbb43696cfacf73e61916d1860ac30b5b2f. [Accessed 2026-10-12].