ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ZNSwap: un-Block your Swap

Shai Bergman *Technion*

Niklas Cassel *Western Digital*

Matias Bjørling *Western Digital*

Mark Silberstein *Technion*

Abstract

We introduce *ZNSwap*, a novel swap subsystem optimized for the recent Zoned Namespace (ZNS) SSDs. ZNSwap leverages ZNS's explicit control over data management on the drive and introduces a space-efficient host-side Garbage Collector (GC) for swap storage co-designed with the OS swap logic. ZNSwap enables cross-layer optimizations, such as direct access to the in-kernel swap usage statistics by the GC to enable fine-grain swap storage management, and correct accounting of the GC bandwidth usage in the OS resource isolation mechanisms to improve performance isolation in multi-tenant environments. We evaluate ZNSwap using standard Linux swap benchmarks and two production key-value stores. ZNSwap shows significant performance benefits over the Linux swap on traditional SSDs, such as stable throughput for different memory access patterns, and 10× lower 99th percentile latency and 5× higher throughput for `memcached` key-value store under realistic usage scenarios.

1 Introduction

Swap is regaining interest from the academia, industry, and kernel communities [2, 3, 12–14, 42, 43, 53, 54] as SSDs are getting faster with both low-latency NAND and high-speed PCIe interfaces [5, 11, 51]. Swap on SSDs is no longer viewed as a last-resort memory-overflow mechanism, but as a crucial system component essential for effective memory reclamation and high system efficiency [3, 14, 18].

Unfortunately, the broader deployment of SSDs as swap devices is overshadowed by several notable performance issues. One of the key limitations is the system performance degradation as the SSD utilization increases. For example, Figure 1 shows a drastic swap bandwidth drop as the device space usage grows beyond 20%, forcing low space utilization to maintain high performance. In § 3 we thoroughly analyze this and other performance issues with swap on SSDs, such as bandwidth variations for different memory access patterns, and poor isolation in a multi-tenant setting.

These performance anomalies have no simple solution. They stem from the inherent mismatch between the easy-to-use block-interface abstraction and the intrinsic flash media

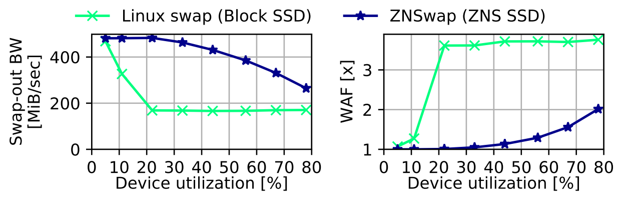

Figure 1: Swap-out bandwidth of random memory accesses (a common swap access pattern [43, 55]), with default Linux swap on Block SSD and ZNSwap on ZNS SSD. The two 1TB SSDs share the same hardware platform and media. WAF–Write Amplification Factor.

properties. In particular, this interface deliberately conceals the absence of in-place updates to flash-based media. Under the hood, updates are written out-of-place to a specifically allocated set of flash blocks (i.e., erase-block). To this end, SSD controllers implement a Flash Translation Layer (FTL), which translates the host-side random writes into sequential writes required by the media, and maintains logic-to-physical mapping for each block. It further entails a device-side Garbage Collection (GC) process to free up erase blocks and reclaim capacity for new writes.

More crucially, this interface decouples the media management from the host-side software stack, so neither the software using the SSD nor the SSD's management logic have any visibility into each other activities. In the context of swap, this decoupling hinders the OS's ability to optimize data placement on the device, and the device's ability to leverage unique characteristics of the swap mechanisms and its usage of the device. For example, the performance degradation observed in Figure 1 is caused by *Write Amplification (WA)*, i.e., the extra data movements performed by the GC. Notably, as we show in § 3, the Write Amplification Factor (WAF) (Figure 1, right) could have been reduced if only the GC were aware of the OS-managed validity status of the stored blocks.

Zoned Storage interface for SSDs (ZNS) [4] aims to reestablish the host's control over key aspects of the storage device management [25]. ZNS opens unique opportunities for cross-layer optimizations that allow novel storage-application co-design simultaneously tailored to the properties of the storage media and its use by applications [25].

At a high level, ZNS introduces the concepts of zones. Zones disallow in-place updates and require their blocks to be written sequentially. To reclaim the space in a zone it needs to be reset, and new writes can be issued. One important benefit of the ZNS interface over prior attempts to expose flash media control to applications (i.e., raw flash or open-channel SSD [26]), is that it enables host-side storage control without having to deal with low-level media management such as wear-leveling or error correction.

In this work, we introduce **ZNSwap**, a novel swap subsystem for Linux that explores the advantages of the synergy between the SSD management and the OS swap logic, leveraging the ZNS interface to overcome the swap performance issues with block-interface SSDs. While prior works observed that the direct application control over SSDs can be beneficial in the context of file-systems and key-value stores [25, 26, 30, 59], ZNSwap is the first to leverage such control for the OS swap on SSDs.

ZNSwap provides a novel, space-efficient host-side mechanism for SSD space reclamation we call *ZNS Garbage Collector* (ZNGC). Unlike the device-side GC of traditional SSDs, ZNGC is tightly integrated with the OS and has direct access to OS data structures which it uses to optimize its operation.

ZNGC design poses a conceptual challenge, however. The space reclamation process naturally involves the migration of logical blocks on the device, without coordinating the block location changes with the applications that own the stored data. This is not a problem for an SSD-side GC because the user-visible Logical Block Addresses (LBA) remain intact. However, applying this solution to the host-side ZNGC would incur unacceptable space overheads in the host, requiring to maintain reverse mapping for every 4KiB block in TB-scale devices. ZNSwap avoids these overheads in the host by storing the reverse mapping information into the logical block metadata being written alongside the swapped-out page contents. The mapping is guaranteed to be correct during the page lifetime.

More specifically, ZNSwap brings the following benefits:

Fine-grain space management. ZNSwap obviates the need for `TRIM` commands, achieving higher performance and better space utilization. The OS uses `TRIMs` to hint to a Block SSD to deallocate specific LBAs, reducing the load on the SSD-side GC. Unfortunately, the use of `TRIMs` have been mostly disabled in the OS swap for their large overheads [35, 39, 50, 54], at the expense of significant bandwidth drop due to the artificial space bloat (§ 3.1.1). In ZNSwap, the ZNGC leverages the direct access to the OS internal page validity structures, without the costly overheads associated with `TRIMs`.

Dynamic ZNGC optimization. ZNSwap dynamically adjusts the number of swapped-in pages that are also stored in the swap device, improving the performance for read-mostly and mixed read-write workloads. The OS keeps a copy of unmodified swapped-in memory pages in the swap device to avoid the swap-out penalty for those pages. The amount of disk space such pages may occupy is statically capped by the OS (50% in Linux, non-configurable). However, this static threshold does not fit all workloads: lower values degrade read-mostly workloads, whereas higher values affect mixed read-write workloads (§ 3.1.2). ZNSwap monitors the WAF and decreases the storage occupancy when necessary by reclaiming the SSD space from swapped-in pages.

Flexible data placement and space reclaim policies. ZNSwap allows easy customization of the disk space management policies to tailor the GC logic to the swap requirements of a specific system. For example, a policy may force co-location of data with similar lifetimes onto the same zone, which was shown to be useful before [28, 34, 44, 56], or achieve better performance isolation by dedicating a separate zone to handle swap from a specific tenant.

Accurate multi-tenancy accounting. As the ZNGC runs on the host, ZNSwap integrates with the cgroup accounting mechanisms to explicitly attribute GC overheads to different tenants, thus improving performance isolation between them.

To summarize, our main contributions are as follows:

- Thorough analysis of traditional Block SSDs' drawbacks when used as swap devices.
- A mechanism to enable ZNS SSDs to serve for swap, without resource-expensive redirection mechanisms in the host, by leveraging logical block metadata for efficient reverse-mapping.
- Custom swap-aware SSD storage management policies which reduce WA, improve performance, and achieve better isolation in multi-tenant environments.
- Extensive evaluation on standard benchmarks and real applications, demonstrating ZNSwap's performance gains, e.g., up to 10× lower 99th percentile latency and 5× higher throughput for `memcached`, with 2.5× lower WAF when compared to traditional swap on Block SSD.

2 Background

OS swap. When a system encounters memory pressure, it selects victim memory pages for eviction to a *swap device*. The OS unmaps the page chosen for eviction from the page-tables and *swaps-out* the page, writing it to the swap device.

Linux divides the space on a swap device into memory-page-sized blocks called *swap-slots*. The OS allocates a new slot for each page being swapped-out. When a page is swapped-in and the utilized swap device capacity is below 50%, Linux keeps the copy of the page both in memory and in the swap. Such pages belong to the *swap-cache*. The OS evicts swap-cache pages without writing them back to the swap. The swap-slot is freed upon the first write to a swap-cache page, and the page is removed from the swap-cache.

Block SSD space management. The SSD's FTL maps Logical Block Addresses (LBAs) to the physical data locations within erase-blocks on the device. An update to a logical block is implemented by writing the new data to a separate erase-block, and then remapping the host-side LBA to the new block, followed by invalidating the old one. To free space for new writes, a *Garbage Collector* (GC), executed by the SSD controller, reclaims the invalidated blocks and consolidates the still-valid blocks from multiple erase-blocks to a new erase-block, and then erases the freed erase-blocks. This operation requires *over-provisioning* (OP) of the flash media in the drive in order to reduce the number of copies during the GC operation.

The device-side GC competes for bandwidth with the user I/O. The relative increase in the amount of data written due to GC vs. the external writes is called a *Write Amplification Factor* (WAF). The smaller the OP, the higher the WAF and the lower the user-visible performance of the device [34].

Zoned Namespace (ZNS) is a new storage interface for SSDs [25]. A ZNS SSD is organized as a set of logically-addressable *zones*. Each zone is physically aligned to an SSD's erase-block size. Reads inside a zone can be random, but writes must be *sequential*. Writing to a zone can be done via the common write command or through the *Zone Append* command. The latter works by the host specifying the zone, and the SSD returning the specific write location upon completion, which allows multiple in-flight requests to the same zone [24] (unlike the write command).

Each zone may be either `Empty` (initial state), `Open` (after the first write) or `Full` (no longer writable). The SSD maintains a write pointer to the next logical block for each `Open` zone. To rewrite a zone, it must be *reset*, which transitions it into an `Empty` state. There is a hardware limit on the number of simultaneously `Open` zones.

3 Motivation

Swap performance is important for data centers. The proliferation of fast flash-based storage revitalized the use of swap as a way to maximize memory utilization and reduce costs. Today, swapping does not serve for sustaining severe memory pressure alone. Rather, swap acts as a memory extension during moderate loads, e.g., to optimize the in-memory balance between file-backed and anonymous memory pages [3].

Thus, the swap performance is becoming increasingly important. Recent works propose to accelerate the swap with dedicated hardware [42]. Linux kernel added optimizations to its memory reclamation mechanism [13]. Alibaba Cloud added a per-cgroup background reclaim mechanism [12] to improve multi-tenancy support in data centers. Facebook introduced swap controls for the cgroupv2 mechanism and used it in the fbtax2 project to improve system efficiency [10].

This trend highlights the importance of swap in modern systems. However, most of the current works focused on the OS logic alone. Here we present a thorough analysis of the Linux swap performance focusing on the interplay between the swap logic and the SSD behavior.

3.1 Performance anomalies of swap on SSDs

3.1.1 GC is not aware of deallocated swap-slots

As shown in Figure 1, the swap bandwidth decreases as the swapped-out data occupies a larger part of the device. In general, this behavior is expected because the GC overheads grow proportionally to the amount of actively updated data. However, the drop should not occur when a device is almost empty (occupied only 10% of its capacity).

The root cause is that the device-side GC is *not aware* that the OS discards some swapped-out pages and *invalidates* their respective swap-slots because the OS does not by default notify the SSD. Therefore, the *actual* occupancy of the swap device is much higher than the one visible to the OS, leading to higher GC overheads.

To cope with this issue, most SSDs implement a `TRIM` command that allows the OS to *hint* to the SSD to reclaim the storage of invalidated swap-slots. However, in practice, popular Linux distributions (e.g., Debian, Ubuntu) disable the use of `TRIM` command for swap [7, 9, 15, 21]. The reasons include `TRIM` dispatching overheads, the long latency of the `TRIM` command, and the complexity of supporting asynchronous `TRIM`s [35, 39, 50, 54, 54].

When `TRIM`s for swap are explicitly enabled, Linux issues the command *once* for a batch (*cluster*) of 512 invalidated swap-slots, to reduce the overheads. Notably, these swap-slots must be *contiguous* in the LBA address space [1].

To see the performance effect of `TRIM`s, we run the same random-write `vm-scalability` benchmark as in Figure 1, but with `TRIM`s enabled (see § 6 for the setup). We measure the swap-out bandwidth and WAF over time as the device is being used to illustrate gradual performance degradation.

Figure 2 shows that `TRIM`s (512-slot) have negligible effect. This is because the LBA contiguity requirement of `TRIM` clusters in Linux effectively prevents issuing `TRIM`s for the majority of the invalidated slots. These results corroborate the note in the `swapon` man page [20] that enabling `TRIM`s often does not improve swap performance.

Finer-grain `TRIM`s are not effective either. To demonstrate this, we develop a special mechanism that enables `TRIM`s for small contiguous clusters of eight swap-slots. This is not a practical approach, however, due to its high overheads (see § 6.1.1) Figure 2 shows slight performance improvement, but still 2× lower than the maximum bandwidth. Clusters smaller than 8 slots result in a prohibitively high rate of `TRIM`s, so the SSD cannot keep up with the swap-slot invalidation rate.

Observation I: `TRIM`*s are not effective at lowering GC overheads for swap.*

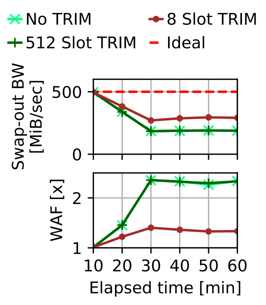

Figure 2: Swap-out bandwidth over time. Random memory writes using 40% of swap capacity.

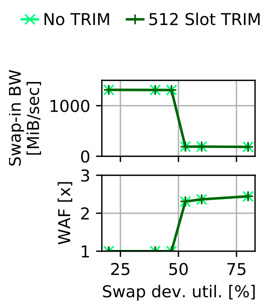

Figure 3: Swap-in bandwidth of random reads as a function of swap capacity utilization.

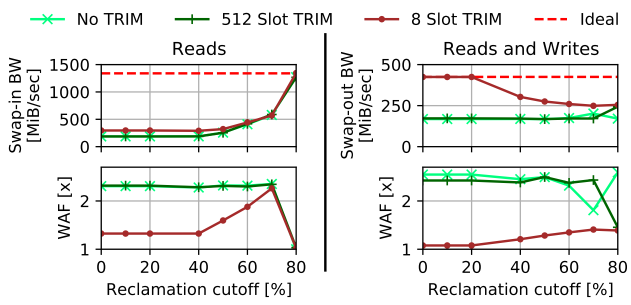

Figure 4: Swap-in and swap-out bandwidth for random reads and mixed reads and writes workloads respectively for different swap reclamation cutoffs.

3.1.2 Swap cache is not aware of GC

We execute the `vm-scalability` benchmark to perform uniform random reads on a large chunk of memory exceeding physical RAM and measure the swap performance for different values of the swap device utilization. Ideally, we expect the read performance to be independent of the utilization. Instead, Figure 3 shows a 6.9× slowdown and 2.5× WAF above 50% occupancy.

Our analysis shows that this problem occurs due to the way Linux implements its swap-cache. Recall that this cache is comprised of pages that are swapped into memory but the OS still maintains a copy in the swap. When the swap device's utilization exceeds 50% – a hard-coded static parameter we call *swap reclamation cutoff*, Linux stops adding newly swapped-in pages to the swap-cache, invalidating their swap-slots immediately. As a result, the swap-out penalty for such pages incurs writing a page to the swap device, rather than discarding them from memory if they were in the swap-cache.

We suggest two possible reasons for this implementation. First, as the swap device gets full, the swap-slot allocation algorithm scans the swap-slot array linearly, which becomes slow [6]. Second, in the context of SSDs, deferring the swap-slot invalidation for in-memory pages effectively increases the device occupancy and eventually reduces performance due to the GC.

Unfortunately, the swap reclamation cutoff establishes a trade-off between the swap-out performance (preferring higher cutoff), and WAF (preferring lower cutoff). To illustrate, we measure the performance of two applications: one performing reads, and the other mixing both reads and writes. This setup aims to show that lower values of the reclamation cutoff are good for write-intensive workloads and bad for read-intensive ones. Higher values mirror this behavior.

We execute `vm-scalability` configured to use 80% of the swap device's capacity. Half of the working set fits in RAM. Figure 4 shows the swap-in and swap-out bandwidth and WAF as a function of the swap reclamation cutoff. For random reads, the swap-in performance increases with the reclamation cutoff, as fewer pages need to be written back upon eviction, with all the pages having copies both in the swap and in memory at the extreme. For the mixed workload, the effect of the cutoff is not visible with the default Linux configuration because the performance is low anyway. But with fine-grain `TRIM`s and higher baseline performance, smaller cutoff values are preferable.

Observation 2: The static reclamation cutoff strives to strike a balance between read and write performance, but instead aggressively prioritizes write workloads when the swap occupancy grows.

3.1.3 GC is not aware of page access pattern

We evaluate the performance of workloads with different memory access patterns using `pmbench`. We consider two write workloads: with uniform and with skewed access distributions (normal, $\sigma = \frac{1}{12}$ of the working set size, the default in `pmbench`). The swap-out bandwidth is 480MiB/sec (maximum for this SSD), and 195MiB/sec (WAF is 2.5) respectively, when using 5% of the swap capacity and 512-slot `TRIM`s enabled.

This difference stems from the different lifetimes of swapped-out pages. With the skewed distribution of memory writes, there are fewer opportunities for the swap subsystem to find large contiguous clusters of swap-slots to perform `TRIM`s, whereas uniformly distributed writes result in the swap-slots of similar lifetimes, increasing the chances of finding such clusters. 8-slot `TRIM`s are better, but the performance is still suboptimal: 324MiB/sec, with WAF of 1.5×.

Observation 3: Swap performance may vary significantly depending on the memory access pattern.

3.1.4 GC is not aware of OS's performance isolation

Linux's cgroup mechanism enforces resource isolation among different processes. In particular, it is possible to isolate swap bandwidth via `blk-throttle`. This is useful, e.g., in container-based virtualized environments to prevent performance interference between containers.

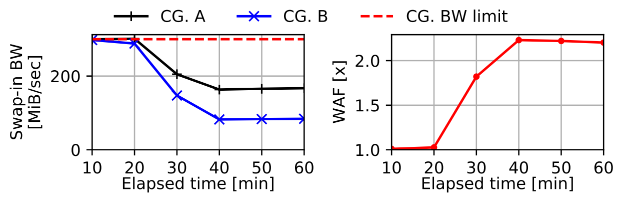

Figure 5: Swap-in bandwidth and WAF of 100%-random-read cgroup (A) and 50/50%-random-read/write cgroup (B) co-running together, each throttled to 300MiB/sec reads and 300MiB/sec writes.

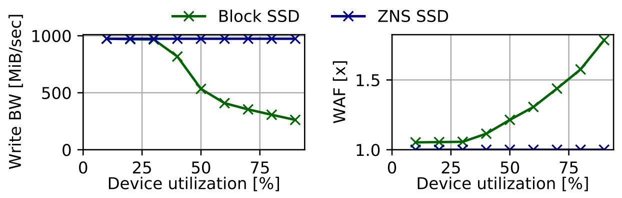

Figure 6: Write bandwidth and WA of sequential writes and TRIM operations to erase-block sized regions on Block SSD and ZNS SSD as a function of device utilization.

We now evaluate the quality of the performance isolation in a scenario where we expect no interference. We run two processes, each in its own cgroup limited to 300MiB/sec reads and 300MiB/sec writes from/to the swap device. One process performs 100% reads and the other executes an equal mix of reads and writes, all uniformly distributed. To prevent any interference the processes are pinned to separate sets of cores, each with its own device queue. The aggregate bandwidth of the SSD does not reach its limit (1GiB/sec).

We expect both processes to achieve their maximum bandwidth allocation. In practice, during the first 20min of the execution (Figure 5) no GC is performed, thus the SSD sustains the cumulative request rate from both processes. When the GC is triggered, the swap-in bandwidth of *both* cgroups drops. Importantly, the first process performs only reads, and should not have been affected by the GC overheads caused only by the writes of the second process. This behavior stems from the GC's inability to distinguish between the I/Os from different processes, and the OS's inability to enforce bandwidth limits on the GC.

Observation 4: The GC impairs performance isolation dictated by the host OS.

3.2 Opportunities with swap on ZNS

ZNS SSDs provide better control over physical data placement, thereby enabling tighter coupling between the application logic and the device management, and have already been shown to offer new optimization opportunities for production Key-Value-Stores [25]. These results motivate a new GC-swap subsystem co-design that can leverage this coupling to mitigate the performance problems of traditional SSDs discussed above.

Is ZNS essential for performance? An important question is whether there is an inherent benefit to using ZNS SSDs over traditional ones, or one can redesign the swap subsystem alone to achieve the same outcome. In other words, can we achieve the performance of ZNS by emulating it on top of a Block SSD?

To answer, we run an experiment on a Block SSD while using a write access pattern that mimics the one enforced by ZNS zones. We run multiple threads, each performing 4KiB logically sequential writes with 1GiB-TRIMs (the size of an erase-block and a ZNS zone on our device). Each thread accesses its own part of a drive, and overwrites the available space, issuing a TRIM for the whole next 1GiB chunk. Multiple threads are used to emulate typical swap behavior.

We run the experiment for different values of device utilization. Figure 6 shows the results. We observe that the performance starts to decrease when a device is 30% full, drops to a half of the maximum bandwidth at 50%, and degrades down to a quarter at 80%. This is expected because the Block SSD cannot ensure that host-side TRIMs are aligned with physical erase-blocks as the writes from different threads get mixed in the device, even though the host strives to align them at the LBA level. In contrast, the same experiment on ZNS drives maintains full bandwidth no matter how full the device is.

We conclude that the *ZNS interface offers unique advantages that cannot be achieved with traditional Block SSDs.*

ZNS adoption. ZNS SSDs are expected to gain broader adoption in the near future. They hold the promise to reduce storage costs as they lower the internal DRAM size requirements, and might help reduce media overprovisioning via application-optimized software stack [25].

While the ZNS interface is not backward compatible with the in-place block interface, there is growing support for ZNS at the file system level. For example, F2FS and Btrfs in Linux can utilize ZNS drives.

These trends motivate us to tailor OS swap for ZNS SSDs.

4 Design

ZNSwap addresses three key design goals.

Resource-efficient Host-side GC. Reclaiming storage space in ZNS SSDs requires a host-side process akin to a GC that consolidates valid blocks from fragmented zones into new ones, subsequently erasing the freed zones. The primary challenge is in minimizing the memory and CPU overheads associated with the host-GC operation. This is because, unlike the device-side GC, the host-side GC directly competes for these host resources with regular applications. In essence, we need to *on-load* the GC onto the CPU from the device with minimal costs, thereby enabling its tighter integration with the swap.

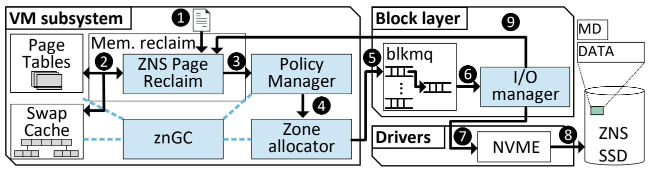

Figure 7: ZNSwap overview. Shaded shapes are internal ZNSwap components.

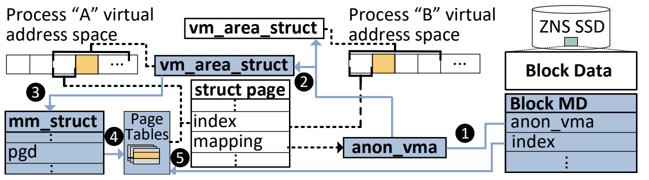

Figure 8: Linux reverse mapping overview. Shaded shapes are data structures accessed during ZNGC reverse mapping.

These resource constraints preclude direct porting of existing host-side GC implementations. In particular, such implementations commonly maintain large translation tables (FTL) [32, 37], which consume about 1GiB for every TiB of data. The tables are frequently updated by writes and GC operations and accessed during reads. Given the typically poor locality of swap-induced I/O accesses [43, 55], these tables have to be resident in host memory. Maintaining the extra level of indirection between logical and physical block addresses appears to be inevitable to allow the host-side GC to move data without affecting the applications using it.

Our host-side GC, *znGC*, eliminates the need for the extra level of indirection entirely. It takes advantage of the fact that the swapped-out pages are still maintained in their owner's page tables, and thus stores the relevant kernel reverse mapping metadata alongside the swapped-out page in the SSD. It also avoids I/O overheads to manage the reverse mappings by using the per-LBA metadata region available in NVMe devices as we describe in §§ 4.2 and 5.1.

ZNGC-OS integration. The key benefit of ZNGC over device-side GC is the ability to access the information exposed by the OS to optimize the swap I/O performance. For example, ZNGC may consult the OS-maintained swap-slot array to identify OS-invalidated swap-slots and avoid redundant copies without using TRIMs. We explain this and additional optimizations in § 4.3.

Swap data placement policies. Swap data placement may have a significant effect on the system performance, but the placement policy may depend on the specific execution environment. For example, a policy to achieve better resource isolation between a pair of processes might prefer storing all the pages of the same process in the same SSD zone. We strive to facilitate the implementation of such policies via a set of APIs that hide the complexity of zone management and ZNGC logic. We explain the API and the policies in § 4.3.

4.1 Overview

Figure 7 shows ZNSwap's main design components. We explain each component and its role using the swap-out path as an example.

After a page to be swapped-out is selected by the OS, it is passed to the *ZNS page reclaim* ❶ which handles the page-table and swap-cache operations ❷. In contrast to the original swap logic, it updates the destination location for the swapped-out page *after* it has been written, as dictated by the ZNS zone-append interface. Before writing a page, the page reclaim module consults the *policy manager* ❸ which determines the destination *zone* and may guide ZNGC to free certain zones on the device. The policy manager incorporates custom policies that can be tailored to specific system requirements. The *zone allocator* ❹ seamlessly handles Full zones and allocates a new zone when necessary.

The page is then submitted to the block layer ❺, which subsequently passes the page to ZNSwap's *I/O manager* ❻. The I/O manager merges zone-append operations whenever possible, and generates an I/O request containing the page's data and reverse mapping information used by ZNGC. Finally, the I/O manager hands off the I/O requests to the NVMe driver ❼ which writes to the ZNS SSD ❽, and updates the reclaim module with the page location on the SSD ❾.

4.2 ZNGC

ZNSwap's reclamation mechanism, ZNGC, is tightly integrated with the kernel virtual memory (VM) subsystem. ZNGC runs as a daemon in the kernel and is triggered either when the number of Empty zones is low, or via explicit requests by the ZNSwap policy.

Contrary to Block SSDs, a page moved by ZNGC is assigned a new host-visible address. Without an additional translation layer, ZNGC must update the page tables holding the original page swap-slot to reflect the new location. To this end, ZNGC stores the relevant reverse-mapping metadata alongside the data in the ZNS SSD's per-LBA metadata region to assist later in updating the page tables. The storage interface (i.e., NVMe) allows to retrieve the metadata together with the respective data block in a single I/O operation. Thus, ZNGC retrieves the metadata to perform the reverse lookup of a given page and then updates the page table(s) that own it.

An important question remains: *which information* needs to be stored in the page metadata to guarantee that the reverse mapping remains correct during its lifetime?

To answer it, we leverage the same main data structures and procedures in the Linux kernel used to implement its reverse mapping scheme (Figure 8).

Background: Linux memory mapping structures. Recall that virtual memory pages in a process' address space belong to virtual memory areas (vmas) that represent virtual memory allocations. vmas belong to a processes' virtual memory address spaces (mm_structs), which hold the page table directory (pgd). The physical page descriptor (struct page) holds metadata enabling the reverse mapping. ZNGC stores the same metadata fields in the logical block's metadata on the SSD.

To locate *all* page table entries associated with a physical page, the reverse mapping procedure accesses the anon_vma ❶ data structure, which is present between each physical page and the virtual memory area (vma) structures that map it[1]. The anon_vma structure holds a list of vmas which may map the page ❷ and accounts for changes to the virtual mappings of the physical page. The physical page's descriptor (struct page) does not not directly account virtual mapping changes, rather, the descriptor holds a pointer to the anon_vma in its mapping field.

The mm_structs of each of the vmas that may map the page are accessed ❸, and their page tables are walked ❹ to locate the page table entries. To calculate the virtual address used to walk the page tables, the index metadata value ❺ along with the vma's start virtual memory address are used. The physical address corresponding to the physical page we have initialized the reverse mapping procedure is located in the last level page table entries and subsequently returned. Since swapped-out pages do not have a valid physical address in their page table entry, ZNGC returns the entries that hold the swapped-out location of the swap-slot we were performing the reverse mapping procedure.

Since the anon_vma structure is freed when there are no more vmas which may map the page and the anon_vma pointer in the struct page does not change, storing the pointer to the anon_vma as well as the mapping's offset (index) within the logical block's metadata on the SSD enables the same functionality as reverse mapping within the kernel.

4.3 ZNGC-swap integration

Physical zone information. Each zone is associated with a map of swap-slots. The map holds information on the use of each swap-slot, and whether it is valid, or swap-cached (similar to Linux's swap_map). This information is used by ZNGC during the space reclamation. Note that a swap-slot can be discarded by the OS and ZNGC becomes immediately aware of the change, without using TRIMs as in Block SSDs. ZNGC may decide to reclaim some zones that are mostly free but hold some of the swap cache pages if it runs out of free storage space, making the swap reclamation cutoff parameter in Linux unnecessary.

Swap-zone abstraction. Active zones that can be used for swap-slot allocation are exposed via a *swap-zone* abstraction. A swap-zone is a virtual entity used to hide the complexity of managing physical SSD zones. Swap-zones are backed by Open zones. When an underlying physical zone transitions to the Full state, it is seamlessly replaced by another Open physical zone. The total number of swap-zones is determined by the limit on the number of Open zones in the device.

[1]anonymous pages and vmas only

Function	Purpose
`void rec_zn(int zn)`	Reclaim specified zone
`void pg_inf(pg_i*, u64 pfn)`	Get page statistics
`void vm_inf(vm_i*, u64 pfn)`	Get information on VMA
`void zn_inf(zn_i*, int zn)`	Get information on zone
`void swap_inf(swap_i*)`	Get ZNSwap statistics

```
typedef struct {
  u64 last_swapout_t;
  u16 access_bit_vec;
  int owner_pid;
  u64 cgroup_id;
} pg_i;
```

```
typedef struct {
  u64 vm_flags;
  u64 size;
  int readahead_win_sz;
  u64 cgroup_id;
} vm_i;
```

```
typedef struct {
  int zone_id;
  int capacity;
  int occupied_slots;
  int invalid_slots;
  int swap_cache_slots;
  int swap_zone_id;
} zn_i;
```

```
typedef struct {
  u64 num_{slots,zns};
  u64 free_{slots,zns};
  u8 zslot_array_sz;
  u32 {high,low}_wmark;
  bool gc_running;
} swap_i;
```

Table 1: ZNGC policy API.

ZNSwap policies. ZNSwap provides an API to facilitate the development of custom data placement and zone reclamation policies. A policy is invoked when the OS swaps-out a page, and its primary goal is to determine which swap-zone the page is written to. If there is a need to reclaim some of the zones, the policy may (asynchronously) invoke ZNGC to do so for a specific set of zones. The policies are implemented in a kernel module. Note that the swap-slot allocation policy operates at the granularity of swap-zones rather than swap-slots to conform to the ZNS interface.

API. A policy receives the page frame number (pfn) of the page being swapped-out and returns the swap_zone_id of the swap-zone where the swap-slot should be allocated. Table 1 lists the functions that can be invoked by the policy.

We define three sample policies:

per-core policy Attempts to assign a swap-zone per-CPU-core. If there are more cores than Open zones, the swap-zones are multiplexed. This mimics Linux's swap-cluster per core policy and reduces contention on swap-zones.
hot/cold policy Utilizes a per-page access history bit-vector maintained by the OS and assigns hot and cold pages to different swap-zones.
cgroup policy Attempts to assign a swap-zone per-cgroup. If more cgroups are available, the swap-zones are multiplexed. If cgroup swap limits are set (max number of swap-slots), the policy will reclaim a zone used by the cgroup whose number of used zones exceeds the limit the most (as zones may contain invalidated swap-slots).

Example policy. We use cgroup policy to illustrate the use of the policy API. When invoked, the policy:

1. Selects the destination swap-zone for the cgroup (using

the `cgroup_id` from `pg_inf()`).

2. If the number of free physical zones is below a predefined low watermark (`swap_inf()`):
   - 2.1. Selects a victim cgroup whose number of utilized physical zones exceed its allocated swap-slot capacity the most.
   - 2.2. Iterates over the cgroup's physical zones (obtained via `swap_zone_id` from `zn_inf()` corresponding to the swap-zone allocation of the cgroup) and selects the zone with the least amount of valid slots.
   - 2.3. Triggers an asynchronous explicit reclaim on the victim zone (`rec_zn()`).
   - 2.4. Repeats the procedure until enough zones have been reclaimed (step 2.1).
3. Returns the destination swap-zone.

cgroup accounting. When a cgroup's swapped-out data is copied during the ZNGC operation, ZNGC's bandwidth is accounted as part of the cgroup's total bandwidth to the device. We do not yet implement the CPU accounting, but this is not critical as ZNGC's CPU overhead is low as we show in § 6.1.1.

4.4 Discussion

ZNSwap introduces the zoned namespace interface to core kernel mechanisms which used to support only traditional block devices. The ZNSwap's design is driven by the fundamental characteristics of ZNS SSDs, that are unattainable with traditional Block SSD, and which dictate the following design choices:

- **Zoned interface:** ZNSwap fully adheres to the zoned storage specification, therefore it inherits the specification's integral benefits. For example, ZNSwap utilizes zone-append operations to enable concurrent writes to sequential-write-only zones, accelerating the swap-out procedure to ZNS SSD by sequentially appending page data.
- **ZNS-related host responsibilities as opportunities:** ZNS requires implementing host-side GC, which present new opportunities for WA mitigation, better utilization of the SSD's capacity for swap-cache pages, and for improving performance isolation.
- **Tight integration of ZNSwap with kernel mechanisms:** utilizing fine granularity information the OS attains per swap-slot enables better synergy between the OS and ZNS SSD.

Hardware limitations. The number of possible destination zones for swapped-out pages in ZNSwap's data placement policies are limited by the number of `Open` zones the ZNS SSD supports, which is device specific. The limit affects the granularity of the policies' classifications. ZNSwap is designed to support ZNS SSDs with varying number of `Open` zones and zone sizes, and abstracts the intricacies of zone management via the swap-zone abstraction (§ 4.3).

ZNSwap also requires the use of the ZNS SSD's per-block metadata (64B). While per-block metadata is currently supported primarily in enterprise NVMe-SSDs, we believe that it will be a common feature among ZNS-SSDs.

5 Implementation

ZNSwap adds support for the zoned-interface model to key kernel mechanisms located in several subsystems. We implemented ZNSwap in Linux 5.12 with 4K LOC[2] (CLOC [8]).

5.1 ZNS page reclaim

Linux's reclamation algorithm is incompatible with the zone-append interface because it assumes that the write location of the swapped-out data is known before the write completion. Specifically, the algorithm uses the swap-slot as the key in the swap-cache for the page currently undergoing reclamation. If a page is accessed while it is being written to the swap device, a page-fault is raised, and the kernel locates the page in the swap-cache using the swap-slot entry to remap it.

ZNSwap redesigns the swap-out mechanism not to rely on the pre-acquired swap-slot entry. The main idea is to utilize the dirty bit of the page's page-table entry to indicate whether the page has been dirtied during the data transfer to the swap device. Write access to such a page does not raise a page-fault since the page is *still mapped* in the page-table. Rather, we check the dirty bit in the page-table when unmapping it. We provide more details in Appendix A.

5.2 ZNGC

We now describe the zone reclamation process in detail. ZNGC first selects a candidate zone from a preselected set of zones supplied by the policy. Given a zone, ZNGC scans through batches of pages until a whole zone is reclaimed. Figure 9 depicts the main stages:

Gather. ZNGC checks the swap-slots in the candidate zone. Swap-slots of the pages currently cached by the swap-cache are removed from the swap-cache and their swap-slots are invalidated. Occupied valid swap-slots are gathered into a pre-allocated array of block IOs to perform device reads. This stage completes when the block IO array is full, or until it reaches the end of the zone.

Read. The occupied blocks IOs containing the read operations are dispatched as a batch of requests to the device. The destination of each read operation corresponds to a page from a pre-allocated page pool. The metadata for each swap-slot is read from the device into a buffer.

Write. Once all read operations are complete, each occupied page from the page pool is examined and assigned a destination zone based on the ZNGC-swap policy. The block IO array is subsequently reused to hold all the pending write requests, which are dispatched as a batch.

[2]https://github.com/acsl-technion/znswap

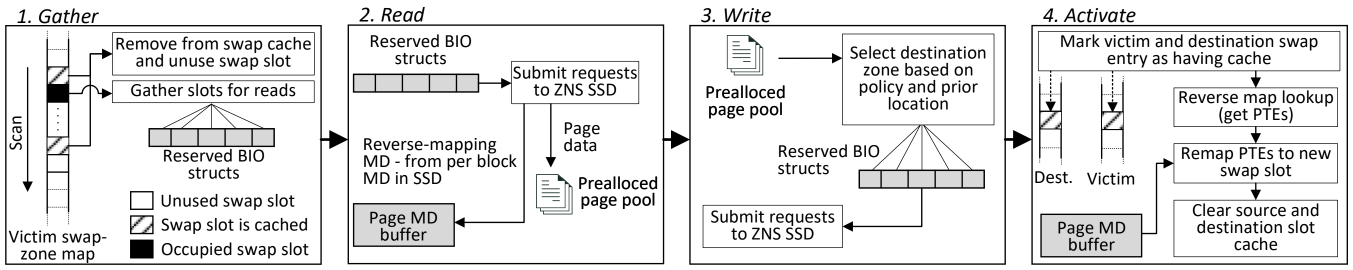
1. Gather
Remove from swap cache and unuse swap slot
Gather slots for reads
Scan
Reserved BIO structs
Unused swap slot
Swap slot is cached
Occupied swap slot
Victim swap-zone map
2. Read
Reserved BIO structs
Submit requests to ZNS SSD
Page data
Reverse-mapping MD - from per block MD in SSD
Preallocated page pool
Page MD buffer
3. Write
Preallocated page pool
Select destination zone based on policy and prior location
Reserved BIO structs
Submit requests to ZNS SSD
4. Activate
Mark victim and destination swap entry as having cache
Reverse map lookup (get PTEs)
Remap PTEs to new swap slot
Clear source and destination slot cache
Dest.
Victim
Page MD buffer

Figure 9: ZNGC: main stages in garbage-collecting a victim zone.

Activate. After the write operations complete, the corresponding swap-slots in the victim's zone are re-examined. If a swap-slot is still valid and occupied, it is marked as if it resides in swap-cache in both victim and destination zones, to indicate to other kernel procedures that these swap-slots are currently in use. The page-table entries corresponding to the victim swap-slots are subsequently remapped to hold the destination swap-slots with the help of the reverse mapping information obtained from the metadata (`mapping` and `index`). Finally, the victim swap-slots are cleared. After ZNGC traversal over a zone is complete, the zone is reset.

Concurrent accesses to swapped-out pages undergoing migration trigger a regular swap-in operation. ZNGC will skip the corresponding swap-slot's migration as the page resides in memory, and will continue with the reclamation of the zone.

ZNGC does not perform dynamic memory allocations and is designed to occupy a minimal amount of physical memory (up to 5MiB).

5.3 I/O manager

ZNSwap's I/O manager adds support for zone-append merges and seamlessly stores the per-page reverse-mapping information into the metadata region of each written LBA.

Zone append merges. ZNGC and the page-out procedures take advantage of the `blk_plug` mechanism to batch together zone-append operations destined to the same zone. We add support for zone-append merges in the block layer by identifying block IOs destined towards the same zone that are waiting to be drained and merge the page-list of each block IO, creating a single block IO request. Once the request has been completed, we iterate over the pages in the request and generate an independent completion notification to each of the merged block IOs with their respective write location, calculated from their offset within the merged block IO and its final location.

Reverse mapping metadata. The I/O manager allocates a DMA-mapped physical page pool for metadata associated operations. The pages serve as a host buffer for the per-page metadata, and act either as a source or target location for append and read I/Os, respectively. The DMA address of the pages is supplied as part of the per-LBA metadata for each command. When serving a swap-out append operation, each LBA stores 16 bytes of metadata for the reverse mapping information of the page (`mapping` and `index`).

6 Evaluation

Our evaluation demonstrates the benefits of ZNSwap using ZNS SSDs over the Linux swap using Block SSDs. In particular, we focus on the benefits of integrating the ZNGC with the host OS and the usefulness of ZNGC policies. We note that all our benchmarks perform direct *memory* accesses only, and impose SSD accesses due to the swap activity. Thus, the actual SSD access pattern might differ from the memory access pattern in the benchmark.

Can ZNS drives be used via compatibility layers? Linux swap does not work on top of ZNS drives, either as a swap-file or a swap partition. Existing Linux device-mappers such as `dm-zoned` [49] and `dm-zap` [33] aim to expose zoned devices as regular block devices without any write-pattern constraints but require large mapping structures for indirection. However, they do not currently support ZNS SSDs. Therefore, the only plausible baseline is Linux swap with block SSDs.

Hardware. We use a server with 2× Intel Xeon Silver 4216 CPU and 512 GiB of memory (2× 256 GiB DDR4 2933 Hz), with Ubuntu 20.04, Linux 5.12.0. HyperThreading is disabled, the frequency governor is "performance", and "turbo" is disabled to achieve stable results. We use a 1 TB production-grade Western Digital ZN540 ZNS SSD and an equivalent 1TB conventional Block SSD (with 7% OP) that uses the same hardware platform and media. Both SSD's maximum sequential read and write bandwidth is 3.2 GiB/sec and 1 GiB/sec respectively. Random 4 KiB reads and writes reach 1.4 GiB/sec and 1 GiB/sec respectively. For the ZNS SSD, the writeable capacity of each zone is 1077 MiB, and is formatted with the ability to store 64 B of metadata per LBA.

Setup. We configure the swap size to be the size of the system memory (512 GiB) according to the common practice [19]. The remaining capacity on the drives is filled with data. The resulting effective OP of the swap partition in the Block SSD is 12%, therefore we configure ZNSwap to use the same OP.

Before each experiment, the SSD is formatted, followed by a ramp-up until the workload has reached its steady state [17]. Bandwidth and WAF measurements are sampled at 10 min

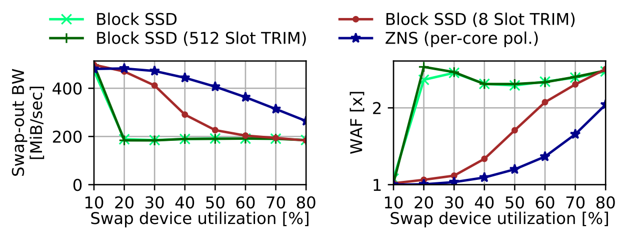

Figure 10: Swap-out bandwidth of `vm-scalability` with random memory writes. As expected, higher device utilization results in higher GC load.

intervals. The Block SSD's WAF is measured through the device's internal host- and media-writes counters, and the ZNS SSD's WAF is measured by recording the number of writes performed by ZNGC.

Performance metrics and optimal performance. We primarily focus on the swap-out bandwidth as the main performance metric. The rationale is that under write-intensive memory access pattern, swap-in operations trigger the eviction of an equal amount of dirty pages to the drive. Hence, the resulting SSD access pattern is an equal mixture of 4KiB random reads and mostly random writes for Block SSD, and random reads and sequential writes for ZNS SSD.

The maximum write bandwidth for such a 50%/50% access pattern on both Block SSD and ZNS SSD drives is measured to be 488 MiB/sec. Therefore, we claim that the ZNSwap's performance benefits over the Linux swap baseline presented in this section *stem primarily from ZNSwap design rather than from the performance differences among the drives or the difference in the access patterns.*

6.1 Synthetic benchmarks

We use the standard swap performance benchmarks, `vm-scalability` [22] and `pmbench` [58] which allow evaluating the performance of the swap subsystem and the swap device under different memory access patterns.

We rerun several experiments from the Motivation section on ZNSwap, to show how it recovers the system performance for the cases where the standard Linux swap on Block SSDs suffers from the performance anomalies.

6.1.1 Benefits of ZNGC-swap subsystem integration

Swapping without `TRIM`s. We execute `vm-scalability` in a 2 GiB memory-limited cgroup. In each experiment, we preallocate different amounts of memory to evaluate different levels of the swap device's capacity utilization. We then perform random writes to that memory (`case-anon-w-rand-mt`), resulting in random read/writes from/to the swap device. To maximize throughput, we execute 64 threads (2× the number of available cores). This is the same experiment as in § 3.1.1.

Figure 10 shows the results. ZNSwap immediately observes the OS-managed swap-slot allocation without using `TRIM`s, and as such only moves the valid pages when running ZNGC. While ZNGC adds overheads to the host, ZNSwap outperforms the Linux swap in all cases but at 10% utilization due to the device WA being lower.

CPU overheads of ZNGC vs. fine-grain `TRIM`s with Block SSD. We measure the maximum CPU overhead of ZNGC under 80% swap device utilization in Figure 10. We observe that ZNGC occupies 15% of a single CPU core. At 10% swap device utilization, the overhead drops to a negligible 0.3%. In contrast, the CPU overhead for dispatching 8-slot `TRIM`s is 32% of a single CPU core with lower swap performance compared to ZNSwap.

Swapping without swap reclamation cutoff. We run the same experiment with read-only and mixed read-write benchmarks as in § 3.1.2 where we established the performance degradation due to the static swap reclamation cutoff in the standard swap. When invoked with ZNSwap, the performance matches the "ideal" line in Figure 4.

6.1.2 Skewed workloads

We run `pmbench` configured to allocate 320 GiB of memory and perform skewed memory writes with the default normal distribution parameters ($\sigma = \frac{1}{12}$ of the allocated memory). The distribution directs 80% of the memory accesses to 20% of the allocated memory considered "hot". The other 80% of the allocated "cold" memory occupies 50% of the swap capacity. The "hot" pages' lifetime in swap tends to be shorter than for other pages. In each experiment, we modify the amount of RAM available to `pmbench` thus varying the proportion of the working set swapped-out from 50% to 90%. This allows to vary the swap device utilization without changing the working set size and page access pattern across the experiments.

We compare the baseline with ZNSwap with the per-core policy that ignores the page access frequency, and ZNSwap with the hot/cold policy that strives to group pages with similar access frequencies into the same zone (see § 4.3).

We make two observations. First, ZNSwap achieves the same performance regardless of the access pattern up to 2× higher bandwidth compared to the baseline for both ZNSwap policies. Second, the hot/cold policy exhibits 15-20% lower WA compared to the naive policy, even though this benefit is not reflected in the swap-out bandwidth in this workload. Reducing WA is important on its own to achieve a higher device lifetime [31].

6.1.3 Swap performance isolation in cgroups

We execute two instances of the `vm-scalability` benchmark, each in a different cgroup. Both cgroup A (CG. A) and cgroup B (CG. B) run with 16 threads. CG. A utilizes 30% of the swap capacity, and performs random *writes*, whereas

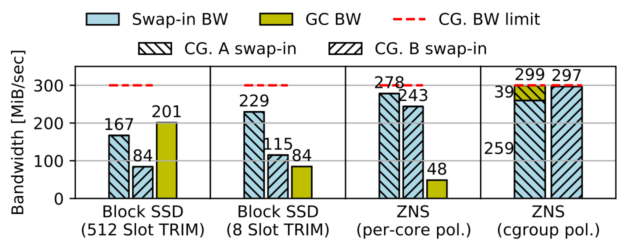

Figure 11: Bandwidth distribution among different cgroups, one reading and another writing data.

CG. B utilizes 10% of the swap capacity and performs only random *reads*. In addition, A's and B's swap bandwidth is limited via `blk-throttle` to 300 MiB/sec. This is the same experiment as in § 3.1.4.

Figure 11 shows the swap-in bandwidth for each cgroup under different configurations. If the swap performance isolation was perfect, each cgroup would behave as if it runs with its own SSD, reaching its target bandwidth (red line). With Block SSD, however, the target bandwidth cannot be attained. Figure 5 shows that without the device-GC overhead, the upper bound is reached, implying that in this experiment this overhead indeed causes performance degradation. Similar to Block SSD, ZNS SSD with ZNSwap's per-core placement policy fails to provide swap isolation.

In contrast, with ZNSwap's cgroup policy, the bandwidth of the writer process (CG. A) fully absorbs the ZNGC bandwidth overheads because only the data attributed to that cgroup is moved by ZNGC. Note that the sum of the bandwidth used by the swap and ZNGC operations in that cgroup does not exceed the predefined limit. CG. B attains full bandwidth, and it is not affected by the ZNGC bandwidth overheads.

6.1.4 Raw swap performance

We stress-test the raw swap-out performance of ZNSwap and its multi-core scaling. We execute `vm-scalability` to sequentially write (`case-anon-w-seq-mt`) 500 GiB of data in a contiguous memory region, while limiting the memory size to 2 GiB via cgroup. By choosing the sequential access pattern, not reusing the same pages, and limiting the number of writes to not surpass the device's capacity, we force the system to avoid reusing swap-slots thus preventing device-side GC and swap-in operations. This is done to achieve the highest performance, stressing the swap software mechanisms.

ZNSwap exhibits the same performance as the traditional Linux swap, achieving 740 MiB/sec swap-out bandwidth for a single core, and the maximum device bandwidth of 1 GiB/sec with 4 cores (no graph shown).

6.2 End-to-end application Benchmarks

We evaluate two popular key-value store servers, demonstrating the benefits of ZNSwap to run large-memory production applications. The throughput and latency we obtain are consistent with those reported for other flash-assisted KVS works [36, 46, 57].

We execute the KV servers on one NUMA node, and the client on the other; hence we set the affinity of both NUMA nodes' `kswapd` threads as well as the `kznsd` thread that executes ZNGC to run on the first NUMA node to co-locate them with the application. Thus, both ZNSwap and traditional Linux swap are allocated *the same* amount of compute resources which they share with the application threads.

6.2.1 `memcached-ETC`

We run a `memcached` key-value store [29] using the `mutilate` client [45] and Facebook's ETC benchmark [23]. We evaluate a random-skewed access pattern with 90% of requests accounting for 10% of the keys. Despite this skewness, the distribution of popular keys in the *memory* is mostly uniformly random because they are scattered across different memory pages. This also dictates random access to the SSD.

We configure `memcached` to use 32 threads on one NUMA node, and invoke 32 `mutilate` client threads on the other NUMA node. We load the data to the server until we reach the target swap device capacity utilization. We do not limit the amount of memory available to the server, thus utilizing all memory (from both NUMA nodes) for the workload. For example, 10% swap utilization (51 GiB) implies the total working set of 563GB. We report the 99p latency of the KV store, maximum throughput, as well as the WAF of the SSD.

Figure 12 shows that ZNSwap consistently outperforms Block SSD-based swap in all performance metrics under the evaluated swap device utilization: under 10% swap device utilization ZNSwap exhibits 10× lower 99p latency and 5× higher maximum QPS while not experiencing any WA, as opposed to Block SSD which suffers from a 2.5× WAF. With the added 8 blk. `TRIM` support for Block SSD, ZNSwap achieves 5× lower 99p, 1.6× higher QPS and 1.1× lower WAF.

6.2.2 `redis-YCSB`

We use an in-memory `redis` data store [16] with the `YCSB` client [27] configured with 50% reads and 50% updates (update-heavy configuration) in a 20-80 hotspot distribution (80% of accesses target 20% of the working set) which is one of the standard options. This memory access pattern induces the same distribution of accesses as we evaluated in § 6.1.2, thereby allowing us to show the application performance impact of the hot/cold placement policy.

`redis` is executed in cluster-mode consisting of 32 servers-threads, running on one NUMA node in a RAM-limited cgroup. It loads a 320 GiB dataset to the cluster. 64 client threads are spawned on the other NUMA node. Similar to the microbenchmark in § 6.1.2, we vary the amount of available RAM while keeping the working set size constant.

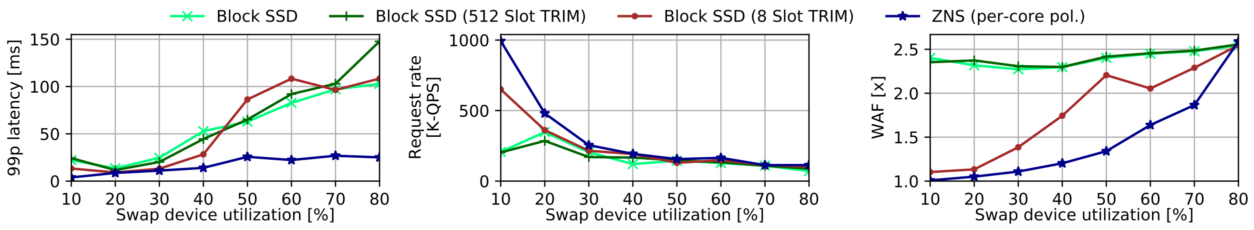

Figure 12: `memcached` Facebook ETC 99 percentile latency at the highest throughput

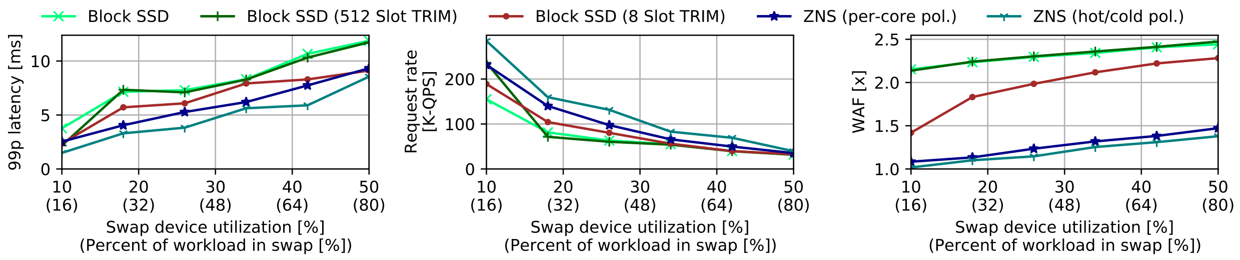

Figure 13: `redis` 20-80 hotspot distribution 50/50 read/write, 99p latency at maximum throughput

Figure 13 shows the 99p latency, throughput and WA for ZNSwap's per-core and hot-cold policies, as well as Block SSD. Both ZNSwap's policies outperform Block SSD in all performance metrics. We observe a 1.27× speedup in throughput and 1.4× drop in latency with 1.1× lower WA for ZNSwap's hot/cold policy compared to the per-core policy.

7 Related work

SSD-friendly swap support. SSDs' unique characteristics warranted a body of works [48, 54] that aim to optimize swap on Block SSDs. These works modify Linux's page reclaim policy (similar to CFLRU [52]) to prioritize reclamation of clean pages and reduce device-side GC overheads without modifying the GC itself. In contrast, ZNSwap offers a novel co-design of the host-side GC and the swap mechanisms and achieves its benefits via tighter coupling between them.

Swap on raw flash. Several early works proposed swap to raw flash [38, 41, 47] thereby avoiding GC overheads due to copying blocks of discarded swap-slots. These papers predated the introduction of native `TRIM` support in SSDs, which was supposed to achieve the same effect. ZNSwap shows that even fine-grain `TRIM`s are not sufficient, and demonstrates other benefits of the tight coupling with the OS enabled by the host-side GC.

Open-channel SSDs [26] expose a low-level storage management interface, similarly to ZNS. ZNSwap's main contribution is its study of the benefits of host-side SSD management and swap co-design, not considered in prior works. Further, unlike ZNS, the adoption of OC-SSDs so far has been limited due to poor portability and the complexity of the host-side media control they require, such as media wear-levelling.

Stream-SSDs [40] expose a traditional block interface, and can reduce WA by utilizing hints so the device may attempt to co-locate data with similar lifetimes onto the same erase-blocks. However, Stream-SSDs' block-interface hinders support for cross-layer optimizations introduced by ZNSwap on ZNS-SSDs, which are key to ZNSwap's performance gains.

Implementing swap data placement support for Stream-SSDs, akin to ZNSwap's swap policies, will offer certain benefits in the scenarios where data lifetime can be predicted and data consolidated into a set number of streams, such as hot/-cold access patterns (as noted in § 6.1.2). However, under random access patterns, Stream-SSDs would perform similarly to traditional Block SSDs. The performance gains pertaining to ZNSwap's cross-layer optimizations that aren't related to data-placement policies (i.e., the elimination of `TRIM`s) exhibit higher performance gains than data-placement policies, as shown in Figure 10.

8 Conclusion

ZNSwap leverages the recent ZNS SSD interface to enable tight integration of the storage management mechanisms with the swap subsystem. ZNSwap introduces a host-side ZNGC that is co-designed with the swap logic to reduce garbage-collection overheads and improve system performance, while also leveraging the tight coupling with the OS and NVMe metadata interface to avoid the costly flash translation layer in the host. ZNSwap demonstrates significant performance advantages of using ZNS for swap in realist scenarios, paving the way to broader adoption of this new technology.

Acknowledgements

We gratefully acknowledge support from Israel Science Foundation (grants 980/21 and 1027/18) and financial support from Western Digital.

References

[1] Swapfile: swap allocation use discard. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7992fde72ce06c73280a1939b7a1e903bc95ef85, 2009.

[2] Making swapping scalable. https://lwn.net/Articles/704478/, 2016.

[3] Reconsidering swapping. https://lwn.net/Articles/690079/, 2016.

[4] NVM Express 2.0 Zoned Namespace Command Set Specification. https://nvmexpress.org/specifications, 2018.

[5] SAMSUNG. Ultra-low latency with Samsung Z-NAND SSD. http://www.samsung.com/us/labs/pdfs/collateral/Samsung_ZNAND_Technology_Brief_v5.pdf, 2019.

[6] Swap: try to scan more free slots even when fragmented. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ed43af10975eef7e21abbb81297d9735448ba4fa, 2020.

[7] Archlinux SSD Optimizations. https://wiki.archlinux.org/title/Solid_state_drive#Continuous_TRIM, 2021.

[8] cloc: Count lines of code. https://github.com/AlDanial/cloc, 2021.

[9] Debian SSD Optimizations. https://wiki.debian.org/SSDOptimization#Mounting_SSD_filesystems, 2021.

[10] Facebook cgroupv2 memory controller. https://facebookmicrosites.github.io/cgroup2/docs/memory-controller.html, 2021.

[11] Kioxia's PCIe 5.0 SSD Just Hit 14,000 MBps. https://www.tomshardware.com/news/kioxia-pcie-5-ssd-just-hit-140000-mbps, 2021.

[12] Memcg backend asynchronous reclaim. https://partners-intl.aliyun.com/help/doc-detail/169535.htm, 2021.

[13] Multi-generational LRU: the next generation. https://lwn.net/Articles/856931/, 2021.

[14] OpenStack: Overcommitting CPU and RAM. https://docs.openstack.org/arch-design/design-compute, 2021.

[15] Red Hat: Discarding Unused Blocks. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/managing_file_systems/discarding-unused-blocks_managing-file-systems, 2021.

[16] Redis. https://redis.io, 2021.

[17] Solid State Storage Performance Test Specification. https://www.snia.org/sites/default/files/technical_work/PTS/SSS_PTS_2.0.2.pdf, 2021.

[18] Swap file on Amazon EC2. https://aws.amazon.com/premiumsupport/knowledge-center/ec2-memory-swap-file/, 2021.

[19] Swap space on Amazon EC2. https://aws.amazon.com/premiumsupport/knowledge-center/ec2-memory-partition-hard-drive/, 2021.

[20] swapon(8) Linux man pages. https://man7.org/linux/man-pages/man8/swapon.8.html,, 2021.

[21] Ubuntu: TRIM the swap partition. https://wiki.ubuntuusers.de/SSD/TRIM/#TRIM-der-Swap-Partition, 2021.

[22] vm-scalability. https://git.kernel.org/pub/scm/linux/kernel/git/wfg/vm-scalability.git, 2021.

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

[24] Matias Bjørling. Zone Append: A New Way of Writing to Zoned Storage. In *Vault Linux Storage and Filesystems Conference*, Santa Clara, CA, February 2020. USENIX Association.

[25] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, DL Moal, G Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*, 2021.

[26] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies FAST 17*, pages 359–374, 2017.

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

[28] Peter Desnoyers. Analytic models of SSD write performance. *ACM Transactions on Storage (TOS)*, 10(2):1–25, 2014.

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

[30] Javier González, Matias Bjørling, Seongno Lee, Charlie Dong, and Yiren Ronnie Huang. Application-driven flash translation layers on open-channel SSDs. In *Proceedings of the 7th non Volatile Memory Workshop (NVMW)*, pages 1–2, 2016.

[31] Laura M Grupp, John D Davis, and Steven Swanson. The bleak future of NAND flash memory. In *FAST*, volume 7, pages 10–2, 2012.

[32] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. *ACM SIGPLAN Notices*, 44(3):229–240, 2009.

[33] Hans Holmberg. dm-zap: Host-based FTL for ZNS SSDs. https://github.com/westerndigitalcorporation/dm-zap, 2021.

[34] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–9, 2009.

[35] Choulseung Hyun, Jongmoo Choi, Donghee Lee, and Sam H Noh. To TRIM or not to TRIM: Judicious trimming for solid state drives. In *Poster presentation in the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[36] Junsu Im, Jinwook Bae, Chanwoo Chung, Sungjin Lee, et al. Pink: High-speed in-storage key-value store with bounded tails. In *2020 USENIX Annual Technical Conference (USENIX ATC' 20)*, pages 173–187, 2020.

[37] Song Jiang, Lei Zhang, XinHao Yuan, Hao Hu, and Yu Chen. S-FTL: An efficient address translation for flash memory by exploiting spatial locality. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2011.

[38] Dawoon Jung, Jin-soo Kim, Seon-yeong Park, Jeong-uk Kang, and Joonwon Lee. Fass: A flash-aware swap system. In *Proc. of International Workshop on Software Support for Portable Storage (IWSSPS)*. Citeseer, 2005.

[39] Dong Hyun Kang and Young Ik Eom. TO FLUSH or NOT: Zero padding in the file system with SSD devices. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–9, 2017.

[40] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *6th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[41] Sohyang Ko, Seonsoo Jun, Yeonseung Ryu, Ohhoon Kwon, and Kern Koh. A new linux swap system for flash memory storage devices. In *2008 International Conference on Computational Sciences and Its Applications*, pages 151–156. IEEE, 2008.

[42] Gyusun Lee, Wenjing Jin, Wonsuk Song, Jeonghun Gong, Jonghyun Bae, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. A case for hardware-based demand paging. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 1103–1116. IEEE, 2020.

[43] Jaehun Lee, Sungmin Park, Minsoo Ryu, and Sooyong Kang. Performance evaluation of the SSD-based swap system for big data processing. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 673–680. IEEE, 2014.

[44] Jongsung Lee and Jin-Soo Kim. An empirical study of hot/cold data separation policies in solid state drives (SSDs). In *Proceedings of the 6th International Systems and Storage Conference*, pages 1–6, 2013.

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

[46] Cheng Li, Hao Chen, Chaoyi Ruan, Xiaosong Ma, and Yinlong Xu. Leveraging NVMe SSDs for building a fast, cost-effective, LSM-tree-based KV Store. *ACM Transactions on Storage (TOS)*, 17(4):1–29, 2021.

[47] Mingwei Lin and Shuyu Chen. Flash-aware linux swap system for portable consumer electronics. *IEEE Transactions on Consumer Electronics*, 58(2):419–427, 2012.

[48] Mingwei Lin, Shuyu Chen, and Guiping Wang. Greedy page replacement algorithm for flash-aware swap system. *IEEE Transactions on Consumer Electronics*, 58(2):435–440, 2012.

[49] Damien Le Moal. dm-zoned: Zoned Block Device device mapper. https://lwn.net/Articles/714387/, 2017.

[50] Trong-Dat Nguyen and Sang-Won Lee. I/O characteristics of MongoDB and trim-based optimization in flash SSDs. In *Proceedings of the Sixth International Conference on Emerging Databases: Technologies, Applications, and Theory*, pages 139–144, 2016.

[51] Ohshima, S. Scaling flash technology to meet application demands. Keynote 3 at Flash Memory Summit 2018, 2018.

[52] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jinsoo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 234–241, 2006.

[53] SeongJae Park, Yunjae Lee, Moonsub Kim, and Heon Y Yeom. Automating context-based access pattern hint injection for system performance and swap storage durability. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[54] Mohit Saxena and Michael M Swift. FlashVM: Virtual Memory Management on Flash. In *USENIX Annual Technical Conference*, 2010.

[55] Taejoon Song, Gunho Lee, and Youngjin Kim. Enhanced flash swap: Using NAND flash as a swap device with lifetime control. In *2019 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–5. IEEE, 2019.

[56] Benny Van Houdt. Performance of garbage collection algorithms for flash-based solid state drives with hot/-cold data. *Performance Evaluation*, 70(10):692–703, 2013.

[57] Shuotao Xu. *Bluecache: A scalable distributed flash-based key-value store*. PhD thesis, Massachusetts Institute of Technology, 2016.

[58] Jisoo Yang and Julian Seymour. Pmbench: A micro-benchmark for profiling paging performance on a system with low-latency SSDs. In *Information Technology-New Generations*, pages 627–633. Springer, 2018.

[59] Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. FlashKV: Accelerating KV performance with open-channel SSDs. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–19, 2017.

A Pageout process

Figure 14 illustrates the operations performed during the pageout process in detail.

Traditional page-out. A candidate anonymous memory page from the inactive-list $\boxed{1}$ is selected to be evicted (not recently accessed $\boxed{2}$) and is not in the swap cache $\boxed{3}$, it is assigned a swap-slot entry $\boxed{4}$. The swap-slot entry is used both as the destination of the page in the swap device, as well as its identifier within the swap-cache. After the page has been inserted into the swap-cache and subsequently unmapped from the page tables $\boxed{6}$, the swap-slot entry value is inserted instead. If the page is dirty $\boxed{7}$, it is unmarked as such, the write operation to the swap device initiates $\boxed{8}$, and the page is reinserted into the head of the inactive list $\boxed{9}$.

After the page has been successfully written to the swap device, it is moved to the tail of the inactive list $\boxed{10}$, where it is then removed for the second time $\boxed{11}$ and passes through the same conditions as in the first iteration. Finally, the page is freed along with its swap-cache entry $\boxed{16}$

If the page is accessed during the write to the swap device, it is located in the swap-cache using the swap-slot entry, and will subsequently fail one of the conditions in $\boxed{12\text{-}15}$.

ZNSwap page-out. Apart from sampling the accessed bit in $\boxed{2}$, the dirty bit is sampled, cleared, and stored in the `PG_dirty` flag of the `struct page`. The page is then assigned a zone per the defined policy $\boxed{4}$ and the append operation to the swap device initiated $\boxed{5}$; the page is then reinserted to the inactive-list $\boxed{6}$. Once the append operation has been completed and the location of the written data retrieved, the page is inserted into the swap-cache. The `PG_dirty` flag is cleared and the page is moved to the tail of the inactive-list $\boxed{7}$. The page then traverses through $\boxed{8\text{-}11}$ and is unmapped from the page tables $\boxed{13}$. If the page has been dirtied since the append operation has initiated $\boxed{14\text{-}15}$, the page-out operation is aborted. The page is finally freed at $\boxed{16}$.

Unlike the traditional page-out algorithm, an access to the page while it undergoing write-back to the swap device will not raise a page-fault and subsequently remapped since it is *still mapped* in the page-tables. Rather, the dirty bit in the page tables is evaluated during the unmapping process $\boxed{14}$, which indicates whether it is safe to free the page or not.

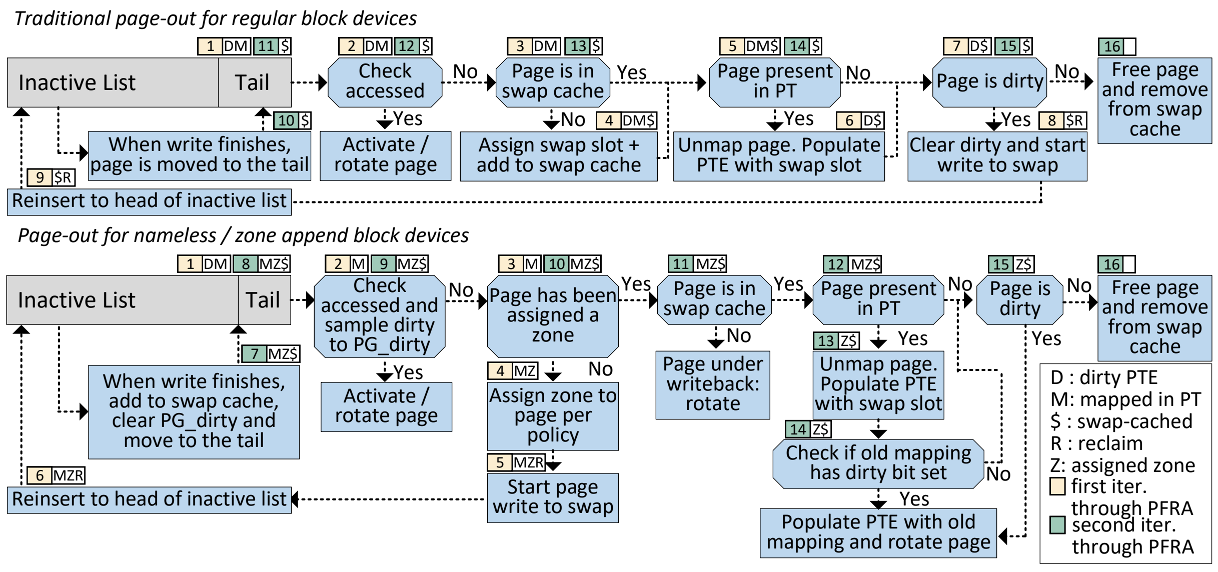
Traditional page-out for regular block devices
Inactive List
Tail
Check accessed
Yes
Activate / rotate page
No
Page is in swap cache
Yes
No
Assign swap slot + add to swap cache
Page present in PT
No
Yes
Unmap page. Populate PTE with swap slot
Page is dirty
No
Yes
Clear dirty and start write to swap
Free page and remove from swap cache
When write finishes, page is moved to the tail
Reinsert to head of inactive list
Page-out for nameless / zone append block devices
Inactive List
Tail
Check accessed and sample dirty to PG_dirty
Yes
Activate / rotate page
No
Page has been assigned a zone
No
Assign zone to page per policy
Start page write to swap
Yes
Page is in swap cache
No
Page under writeback: rotate
Yes
Page present in PT
Yes
Unmap page. Populate PTE with swap slot
Check if old mapping has dirty bit set
Yes
Populate PTE with old mapping and rotate page
No
Page is dirty
No
Yes
Free page and remove from swap cache
When write finishes, add to swap cache, clear PG_dirty and move to the tail
Reinsert to head of inactive list
D : dirty PTE
M: mapped in PT
$: swap-cached
R : reclaim
Z: assigned zone
first iter. through PFRA
second iter. through PFRA

Figure 14: Page-out procedure for inactive pages