NVSwap: Latency-Aware Paging using Non-Volatile Main Memory

Yekang Wu
School of Engineering and Computer Science
Washington State University Vancouver
Vancouver, WA 98685 USA
yekang.wu@wsu.edu

Xuechen Zhang
School of Engineering and Computer Science
Washington State University Vancouver
Vancouver, WA 98685 USA
xuechen.zhang@wsu.edu

***Abstract*—Page relocation (paging) from DRAM to swap devices is an important task of a virtual memory system in operating systems. Existing Linux paging mechanisms have two main deficiencies: (1) they may incur a high I/O latency due to write interference on solid-state disks and aggressive memory page reclaiming rate under high memory pressure and (2) they do not provide predictable latency bound for latency-sensitive applications because they cannot control the allocation of system resources among concurrent processes sharing swap devices.**

In this paper, we present the design and implementation of a latency-aware paging mechanism called NVSwap. It supports a hybrid swap space using both regular secondary storage devices (e.g., solid-state disks) and non-volatile main memory (NVMM). The design is more cost-effective than using only NVMM as swap spaces. Furthermore, NVSwap uses NVMM as a persistent paging buffer to serve the page-out requests and hide the latency of paging between the regular swap device and DRAM. It supports in-situ paging for pages in the persistent paging buffer avoiding the slow I/O path. Finally, NVSwap allows users to specify latency bounds for individual processes or a group of related processes and enforces the bounds by dynamically controlling the resource allocation of NVMM and page reclaiming rate in memory among scheduling units. We have implemented a prototype of NVSwap in the Linux kernel-4.4.241 based on Intel Optane DIMMs. Our results demonstrate that NVSwap reduces paging latency by up to 99% and provides performance guarantee and isolation among concurrent applications sharing swap devices.

***Index Terms*—Paging, Virtual Memory, Storage QoS, Non-Volatile Main Memory**

I. Introduction

In the Linux operating system, paging is designed to extend the main memory capacity using the space of secondary storage devices [1]. The existing paging policy in Linux is designed to improve the overall I/O throughput of concurrent paging workloads. For example, upon paging out, memory pages are written out to swap space in the unit of page clusters [2] to exploit spatial locality. However, the tail latency (i.e., X^{th} *percentile latency*) of paging for a particular application can be unprohibitedly high because it is affected by many factors such as queuing time in kernels and I/O interference of applications concurrently accessing the swap devices. In addition, the latency is unpredictable because the existing paging systems cannot enforce the latency bound of paging for an application, which may result in poor swap experience of users of latency-sensitive applications, e.g., in-memory databases and mobile applications.

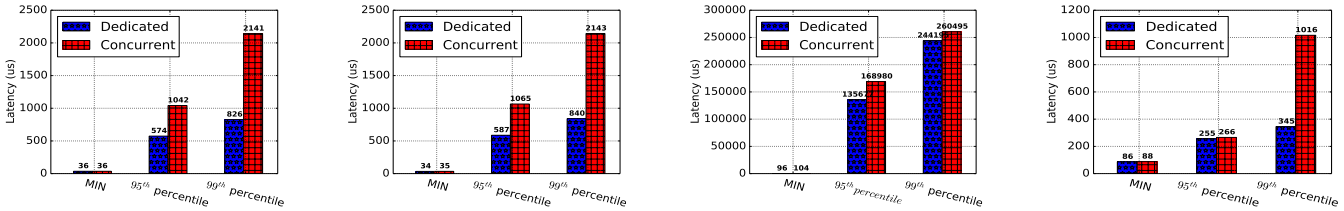
In this paper, we present a new paging system called NVSwap using *Non-Volatile Main Memory* (NVMM) (e.g., Intel Optane DIMMs [3], [4]) to extend memory capacity for serving latency-sensitive memory-demanding applications. Supporting paging to NVMM in operating systems hides the complexity of programming and enables programmers to directly run memory-demanding legacy code without the need of understanding NVMM memory models and their programming interface. NVSwap has the following desirable properties to hide paging latency and improve users' experience.

A cost-effective hybrid swap space: For paging, NVSwap employs two swap zones including a swap zone in NVMM on the memory bus and a regular swap zone in solid-state disks (SSDs) on the I/O bus. Latency-sensitive applications can page to both of the zones according to their latency requirements. Other applications can page to the regular zone for a low cost. NVSwap may swap in pages in NVMM in an in-situ manner (in-situ paging) without extra memory copy, which will further reduce the latency of paging.

Latency-bound enforcement: NVSwap allows users to specify QoS requirements in the form of tail latency bounds of page-in requests for any latency-sensitive processes. These may be set for individual processes or collectively for a group of related processes. NVSwap enforces the latency bound for page-in requests by controlling the space allocation of NVMM among scheduling units (e.g., processes). According to the latency requirements, it also dynamically selects the host swap device from the regular swap zone and the NVMM zone and adjusts the rate of memory page reclaiming to reduce the queuing time of paging requests in the disk scheduling queue.

Persistent paging buffer: We implemented a persistent paging buffer in NVMM for latency enforcement and reducing I/O interference in the regular swap zone. It organizes its space into multiple latency groups, each of which consists of pages from processes having the same latency bound. The pages in the buffer are destaged in the unit of latency groups and in descending order of their respective latency bounds. In this way, pages from latency-sensitive processes will have a higher chance of staying in NVMM. It exploits

978-1-7281-7744-1/21/$31.00 ©2021 IEEE



(a) DB Update Records (b) DB Read Records (c) OS Page Out (d) OS Page In

Fig. 1. (a) and (b) show the minimum, 95^{th} percentile, and 99^{th} percentile latency of DB update and read operations of with one instance of memcached ($Dedicated$) and two concurrent instances ($Concurrent$). (c) and (d) show the minimum, 95^{th} percentile, and 99^{th} percentile latency of page-out and page-in operations during the execution of memcached.

the temporal locality by storing the pages in the same group in the order of eviction from DRAM. By serving the page-out requests using the persistent paging buffers and providing higher priority to read requests in the disk scheduling queue, NVSwap significantly reduces write interference on SSDs. Finally, subsequent page-in requests that hit any pages in the buffer will be directly copied to the swap zone in NVMM and then mapped to process address spaces. With the help of in-situ paging, the page can be immediately used by the process without triggering the overhead of block-level I/O processing.

We have implemented a prototype of NVSwap in the Linux kernel-4.4.241. Our extensive evaluation with the in-memory database and YCSB benchmark show that NVSwap can reduce the paging latency by up to 99% compared to those using only SSDs for swapping. Furthermore, it dynamically adapts the allocation of system resources for enforcing the X^{th} percentile latency for concurrent paging workloads and provides the desired performance isolation among them.

II. Motivation and Related Work

A. Needs for Latency-Aware Paging

Paging in virtual memory is a core component of the Linux operating system. Paging out happens when the swap out daemon $kswapd$ in the kernel is wakened up under memory pressure or when direct page reclaiming is required under an even higher memory pressure [5], [6]. To swap out a page in DRAM, the kernel needs to generate a block-level I/O request and adds it to a disk scheduling queue associated with the device for dispatch. Then the page is written to the swap space hosted on block storage devices, e.g., SSDs. When a page fault is triggered by STORE or LOAD CPU instructions, the kernel needs to swap in the page to be accessed back to DRAM. Paging to NVMM hides the complexity of the new memory models and enables programmers to smoothly adapt their applications to the new hardware.

The current Linux paging mechanism is designed to improve the overall I/O bandwidth of slow swap devices [2] by exploiting spatial locality. For paging out, the kernel typically selects 32 pages from the list of inactive pages and sequentially writes them in a page cluster on swap devices. The size of a cluster ranges from 8 KB to 4096 KB. For paging in, the kernel prefetches multiple pages in a cluster benefiting from high sequential read bandwidth of storage devices.

However, these design options may cause a long paging latency at both kernel and application levels. To illustrate the impact of paging on the latency of major operations of applications. We run the $memcached$ in-memory database server provided in the $YCSB$ benchmark [7]. The server daemons access an SSD-based swap device. The size of the main memory and swap space is set to 5 GB and 10 GB respectively. We run Workload A with 50/50 read/update ratio, 1 KB record size, and zipfian request distribution. The detailed hardware configuration can be found in Section VI. In Figure 1, we compare the latency of major database operations (e.g., update and read) with one $memcached$ server using the swap space ($Dedicated$) to that with two $memcached$ servers concurrently accessing the space ($Concurrent$). We have the following observations from the results.

1. During paging, both of the DB read and update operations may have a long tail latency. With dedicated accesses to the swap space, the 99^{th} percentile latency of DB read operations (840 us as shown in Figure 1(b)) is 25X longer than its minimum latency (34 us). With concurrent accesses to the swap space, the 99^{th} percentile latency of read operations is increased to 2143 us which is 61X higher than its minimum latency (35 us) while the minimum latency is increased by only 3%. A similar pattern is observed for DB updates. This is because the read (page-in) requests might be blocked by write (page-out) requests, causing a long tail latency [8] at the application level when page-in requests and page-out requests are mixed and concurrently access the SSD.

2. The Linux paging system is not able to enforce latency bounds. We observed the huge variation between minimum latency and its corresponding 99^{th} percentile latency for page-in and page-out requests in Figure 1.

3. The OS page-in latency has a critical impact on the latency of DB operations at the application level. Our experimental results show that the OS page-in latency is directly correlated to users' perceived latency because the latency of serving page-in requests is in the critical path of major page faults while page-out requests can be asynchronously served in the kernel. As a result, the 99^{th} percentile latency of page-in requests is comparable to those of DB reads and updates. In contrast, the tail latency of page-out requests can be 122X higher than the latency of the DB operations.

In summary, the current deficiencies in the design of the

Linux paging system prevent users who are sensitive to latency from using the swap space and prevent paging from being used with in-memory applications and mobile devices requiring predictable and low latency bounds for improving users' experience or used with high-performance computing applications whose performance is sensitive to OS noises.

B. Related Work

Paging Approaches for NVMM: NVMM is used for paging because it has low latency and vendor-guaranteed lifespan [3], [9] of 5 years at a minimum. Memorage manages NVMM as storage space when storage capacity is low and manages it as the main memory extension when the availability of memory pages is low [10]. By dynamically changing the allocation of NVMM to main memory, it uses existing virtual memory managers to improve the performance of in-memory applications. Dr. Swap uses a direct read to reduce the overhead of memory copy from NVMM to DRAM for paging in [11]. Refinery swap and nCode were designed to reduce the number of page-out and page-in requests by swapping out less-frequently accessed pages [12] or read-only code page to NVMM [13]. Both SmartSwap [14] and Mars [15] reduce the user-perceived latency of application relaunching in mobile devices. They typically swap in the whole process address space of the application for relaunching. Other paging approaches have been designed to reduce paging overhead in virtual machines using distributed NVMM [16]. Awad et al. comprehensively studied the impact of the existing techniques (e.g., page prefetching, and page replacement algorithms) on the performance of NVMM-based paging systems [17].

Latency Enforcement in Storage Systems: For non-distributed storage systems, SARC+Avatar is a two-level scheduler that uses the earliest deadline first scheduling policy to achieve latency control [18]. In the Xen hypervisor, PSLO was proposed to enforce tail latency for consolidated VM storage by controlling the level of I/O concurrency and arrival rate for each VM issue queue [19]. Tiny-tail flash was designed to eliminate the tail latency induced by garbage collection in solid-state disks. Because the Linux paging system does not maintain an individual issue queue for each process, NVSwap cannot directly use the approaches like AVATAR or PSLO. Instead, it uses user-defined latency bound to control the disk scheduling queue size and process paging location. It also needs to adjust the page pre-cleaning rate and the arrival rate of page-out requests of processes competing for the slots of the disk scheduling queue and NVMM space. In parallel storage systems, Zhang et al. provides QoS support for I/O-intesive applications [20]. Karki et al. designed scheduling algorithms to provide QoS enforcement in I/O pipelines [21].

III. NVSwap Design

The objective of NVSwap is to enforce latency bounds of paging for latency-sensitive processes using NVMM. We focus on page-in latency in the paper because it has a direct impact on users' perceived latency as shown in Section II. In this section, we discuss the key concepts and overall design

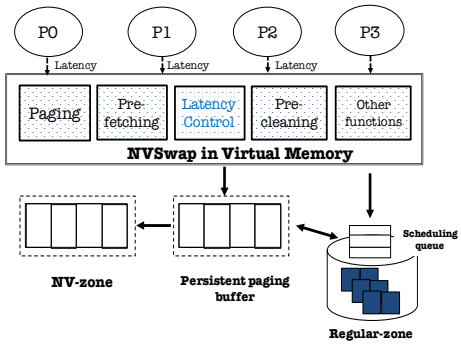


Fig. 2. NVSwap system architecture. The latency control module is a new module added to the Linux kernel although other modules have also been modified accordingly.

of NVSwap. Figure 2 shows the system architecture with multiple processes accessing a shared swap space. From the perspective of software architecture, NVSwap has similar basic functionalities as Linux including paging to/from disks, page prefetching to hide disk access latency, page pre-cleaning to eagerly swapping out dirty pages before new pages are needed using $kswapd$, and other functionalities (e.g., swap space management). Besides these, NVSwap supports paging to/from NVMM. It has a new *latency control* module, which is responsible for determining memory page reclaim rate and the dynamic allocation of NVMM for each paging process according to its user-specified tail latency bound. Since page-out requests are served asynchronously, we only provide latency control for page-in requests. We describe the algorithm used for latency control and page reclaim in Section IV.

The swap space of NVSwap has four main components: a *regular-zone*, an *NV-zone*, a *persistent paging buffer*, and a *shadow mapping table*. The regular-zone is hosted on block storage devices, e.g., solid-state disks. It is used to serve paging requests dispatched from a disk scheduling queue as the Linux paging scheme does. The NV-zone and persistent paging buffer are hosted in NVMM. They are used to serve paging requests to enforce the latency bounds as specified by users. The NV-zone consists of NVMM page frames that can be directly accessed in process address spaces. The persistent paging buffer stores swapped-out pages from latency-sensitive processes and prefetched pages from the regular-zone. When the buffer is full, NVSwap needs to asynchronously flush pages to the regular-zone in background. When page flushing happens, NVSwap does not need to change the page table entry of its corresponding process. Instead, the new disk location of the flushed pages in the regular-zone is recorded in the shadow mapping table. Then the incoming page-in requests to access the flushed pages will be served using their new disk addresses looked up from the shadow mapping table.

Paging-out: According to the output of the latency control module, the page-out requests are directed to access either the persistent paging buffer or the regular-zone. In Figure 3(a), we illustrate three page-out paths of NVSwap. (1) For $Page(1)$, it is paged out to the persistent paging buffer first and then

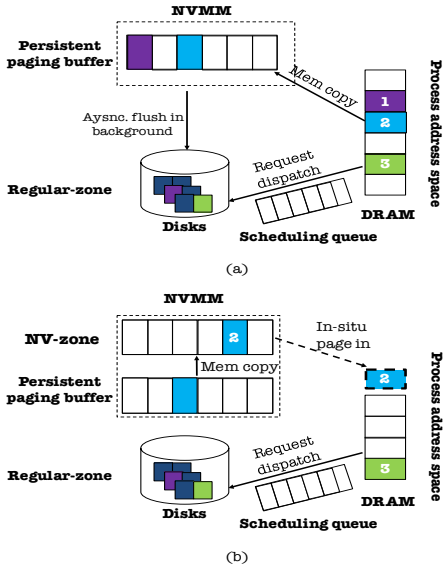


Fig. 3. Illustration of NVSwap paging scheme. (a) Paging out; (b) Paging in.

asynchronously flushed to the regular-zone when the buffer is full. Because the persistent paging buffer is on the memory bus, NVSwap simply copies the page to be swapped out in DRAM to a new page frame in NVMM. Since the persistent paging buffer is non-volatile, a page-out request can be considered complete once it is sent to the main memory extension. We schedule writing pages from NVMM to the regular-zone when the scheduling queue is not saturated. (2) For $Page(2)$, it is simply paged out to NVMM. After the termination of the process referencing $Page(2)$, the page frame is freed for future usage by other processes. (3) $Page(3)$ is paged out to the regular-zone. The page-out request should be dispatched by the scheduling queue in DRAM. The size of the scheduling queue is measured in queue slots. It has a huge impact on the tail latency of paging requests as shown in Figure 1. Therefore, the latency control module periodically adjusts the queue size based on the latency requirements of processes.

Paging-in: When a STORE/LOAD instruction triggers a page fault to access a page in the swap space, NVSwap has two paths for serving the page-in request. We illustrate them in Figure 3(b). (1) If the page (e.g., $Page(3)$) is stored in the regular-zone, NVSwap first issues a read request to read the page from the block device to a new DRAM page frame allocated for serving the page fault. Then by updating the page table entry (PTE) of the process which references the page, it sets up the PTE mapping from the virtual address to the physical address in DRAM. Finally, the process can write/read the data to/from the page. This page-in path is the same as Linux. (2) If the page (e.g., $Page(2)$) is stored in the persistent paging buffer, NVSwap allocates a page frame in the NV-zone. Then it sets up the PTE mapping from the virtual address to the physical address in NVMM. Finally, it copies the data from the persistent paging buffer to the NVMM page frame in the NV-zone. The existing buffer slot hosting the page is freed. This operation is called *in-situ paging* in NVSwap. In-situ paging replaces the operation of reading the regular-zone with the memory copy from the persistent paging buffer to the NV-zone. Consequently, it reduces the page-in latency of serving the page fault.

Resizing the persistent paging buffer: The size of the persistent paging buffer is periodically adjusted according to the ratio of the number of page-in requests and page-out requests. Specifically, let's assume the size of NVMM is C_{nvmm}, the size of the persistent paging buffer is C_{buffer}, and the size of the NV-zone is C_{nvzone}. We further assume the rate of page-in and page-out is $Rate_{in}$ and $Rate_{out}$ respectively. Then $C_{nvzone} = Rate_{in} * C_{nvmm}/(Rate_{out} + Rate_{in})$ and $C_{buffer} = C_{nvmm} - C_{nvzone}$. To calculate $Rate_{in}$ and $Rate_{out}$, NVSwap maintains a moving average of the total number of page-in and page-out requests being served in a 1-second time window. It does not induce additional overhead as Linux already tracks these metrics (e.g., the number of page faults). When the NV-zone is full, in-situ paging is disabled until the existing page frames are freed or more NVMM page frames are allocated for the zone.

The page layout of swap space and prefetching: When flushing occurs in the persistent paging buffer, NVSwap evicts pages from processes that have the highest latency bounds specified by users. For this purpose, it organizes the space of the persistent paging buffer into multiple latency groups, each of which consists of 64 pages from processes having the same latency bound. When a latency group is full, it is split into two latency groups of the same latency requirement. Furthermore, it exploits the temporal locality by storing the pages in the same group in the order of eviction from DRAM. We schedule writing a group of pages from the persistent paging buffer to the regular-zone when the scheduling queue is not full. NVSwap manages page slots using a cluster-based approach like Linux for the regular-zone.

Serving the read requests from the regular-zone is in the critical path and may significantly increase the latency of page fault handling. Linux prefetches pages after a page fault to hide the latency [2]. However, the existing prefetching mechanism reads pages from the regular-zone into DRAM, which may cause memory thrashing under high memory pressure. In contrast, upon page fault to the regular-zone, NVSwap prefetches the pages from the regular-zone to the persistent paging buffer in the unit of a latency group. Furthermore, it only prefetches the pages of latency-sensitive processes that are set to access NVMM according to the output of the latency control module. When page prefetching happens, NVSwap does not need to change the page table entry of its corresponding process. Instead, the new page frame ID of the prefetched pages in NVMM is recorded in the shadow mapping table. The incoming page-in requests to access the prefetched page will be served using the new NVMM page frame. Because our flushing and prefetching algorithm exploits applications' semantics (e.g., latency bound) and temporal locality, the pages in the same cluster in the regular-zone will

likely be accessed together.

IV. Latency Control Module

NVSwap supports storage QoS specified using X^{th} percentile page-in latency. These may be set for individual processes or collectively for a group of related processes. According to our observations, the paging latency is affected by the characteristics of both swap devices and workloads, e.g., read/write latency, disk scheduling queue size, and I/O arrival rate. NVSwap selects a host swap device for each latency-sensitive process according to its tail latency bound. Then according to the latency requirements of the processes accessing the regular-zone, it determines the disk scheduling queue size to control the queuing time. Finally, according to the size of the queue, it adjusts the rate of memory page reclaiming to control the I/O arrival rate. In this section, we describe the algorithm used in NVSwap to enforce the latency bounds.

Selecting the host swap device and the queue size: The default host swap device is the regular-zone for processes. Then given the capacity of the regular-zone, NVSwap may select NVMM as the host swap device for latency-sensitive processes. We adopt a control strategy to estimate the capacity in terms of the scheduling queue size. The strategy is inspired by those used in Storage Resource Pools [22] and PARDA [23].

Let's assume that the paging latency is Lat_i for process P_i (1≤i≤n). Then the latency goal to achieve using the scheduling queue Lat_{goal} is $min(Lat_1, ..., Lat_n)$. The queue size is adjusted based on Lat_{goal} and observed latency $Lat_{observed}$ using Equation 1, where $S(t)$ denotes the size of the scheduling queue at time t and γ is a smoothing parameter between 0 and 1. For measuring $Lat_{observed}$, we instrumented the Linux kernel to collect the latency of paging requests.

$$S(t+1) = (1-\gamma) * S(t) + \gamma * (S(t) * \frac{Lat_{goal}}{Lat_{observed}}) \quad (1)$$

Using the control strategy, if the observed latency is higher than Lat_{goal}, NVSwap will reduce the queue size. Otherwise, it will increase the queue size. If the queue size is too large, we are at risk of losing data in the queue upon system failures. Consequently, we set the maximum queue size to be no larger than S_{max}. If $S(t+1) > S_{max}$, $S(t+1) = S_{max}$. We set S_{max} to be 1024 in the paper. Furthermore, we also set the minimum queue size to be no smaller than S_{min}. For example, S_{min} can be set as the number of channels of SSDs to explore its I/O parallelism. If the queue size $S(t+1)$ is smaller than S_{min} using Equation 1, NVSwap considers that the regular-zone is under-provisioned. It will serve the requests from the most latency-sensitive processes using the persistent write buffer to reduce the load on the regular-zone until $S(t+1)$ becomes no smaller than S_{min}. Algorithm 1 describes the algorithm for the assignment of host swap devices and the determination of queue size.

NVSwap reserves a fixed number of slots in the queue to serve other processes that are not latency-sensitive for solving the starvation issue in request scheduling (#L12). Finally, it is designed to reduce the write interference in the regular-zone. For this purpose, in the scheduling queue, we set read requests to have higher priority than write requests to avoid write interference.

Algorithm 1: Algorithm for the assignment of host swap devices and determination of queue size

Input:
Lat_i: User-specified paging latency of process i, 1≤i≤n;
Set LS: Ordered set $\{ls_1, ls_2, ..., ls_n\}$ of elements from set $\{Lat_1, Lat_2, ..., Lat_n\}$;
index[i]: equals k if ls_i is Lat_k;
S_{min} and S_{max}: minimum and maximum size of the scheduling queue respectively;
$S_{reserve}$: the reserved slots in the scheduling queue;
$Lat_{observed}$: observed latency of accessing the queue.
Output:
Set NS: the set of processes using the persistent paging buffer;
Set RS: the set of processes using the regular-zone;
S(t+1): the size of the scheduling queue.

```
1  NS = {}, RS={1, ..., n}.
2  for k in 1, ..., n do
       // Set the latency goal using the
          minimum latency
3      Lat_goal = ls_k;
       // Update the scheduling queue size
          using user-specified latency
4      S(t+1) = (1-γ)*S(t) + γ*(S(t)*Lat_goal/Lat_observed);
       // Handle the case of
          under-provisioned regular-zone by
          serving latency-sensitive processes
          using persistent paging buffer
5      if S(t+1) < S_min then
6          NS = NS ∪ index[k];
7          RS = RS - index[k];
8      else
9          break;
   // Handle the case of extremely large
      queue size
10 if S(t+1) > S_max then
11     S(t+1) = S_max;
   // Add reserved slots for processes that
      are not latency-sensitive
12 S(t+1) += S_reserve;
```

Latency-sensitive page reclaiming: Page replacement algorithm determines which pages should be swapped out. And *SWAP_CLUSTER_MAX* determines how many pages should be swapped out [24]. It is set to 32 in Linux [2], indicating that $kswapd$ will swap out 32 pages from the list of inactive pages. For latency enforcement, instead of using

SWAP_CLUSTER_MAX with a fixed value, NVSwap sets the maximum number of pages to swap out according to the size of the scheduling queue $S(t)$. As a result, the rate of page scanning matches the capacity of the regular-zone given the latency bounds of processes.

The page replacement algorithm is modified to evict pages from latency-sensitive processes in NS to the persistent paging buffer and evict pages from not in NS to the regular-zone. Specifically, for selecting a page to reclaim, the algorithm scans pages from the end of the *inactive_list* or until the list is empty. We use reverse mapping to map the page frame to its associated process indexed by the process ID. If the process is in NS, NVSwap directs the paging request to access the persistent write buffer. Otherwise, it directs the request to access the regular-zone. The scanning process in a loop is completed until the number of reclaimed pages from processes not in NS reaches $S(t)$ or until the list is empty.

Tail latency monitoring and enforcement: In the Linux kernel, we implemented a monitor, which collects the rate of paging and the X^{th} percentile latency of paging processes for any time window k ($k > 0$). Let's assume that the X^{th} latency specified by users is $Tail_i^k$ for process P_i ($1 \leq i \leq n$) at the time window k. If the observed tail latency is higher than $Tail_i^k$, all the page-out requests from P_i at the next time window $k+1$ will be served using the persistent write buffer. In addition, it will trigger prefetching the pages from P_i so that the page-in requests issued at the time window $k+1$ will be served using NVMM to reduce page-in latency $Tail_i^{k+1}$.

V. Implementation Issues

Admission control: A key question that arises in the implementation of NVSwap is *how many latency-sensitive paging processes can we serve on a hybrid swap space using NVMM?* We need to understand both the system capacity and the total I/O demand to answer the question. We use the following equation to provide a general understanding of I/O demand.

$$DemandIOPS = \sum_{i=1}^{n} \frac{1}{Lat_i} \tag{2}$$

For the capacity of the regular-zone, we suggest computing its throughput (IOPS) using random I/O workloads. The request size should be equal to the page size in Linux, e.g., 4 KB. The measurement should be conducted with an increased number of I/O concurrency. This can be done either during system installation or later by running micro-benchmarks, e.g., IOMETER [25]. NVSwap only copies the pages to the persistent write buffer. For measuring the capacity of the persistent write buffer, we develop a simple tool to measure the latency of copying pages from DRAM to the buffer. Then we convert it to throughput. Using this approach, we obtained the capacity of the regular-zone and persistent write buffer is 7,900 and 215,000 paging I/O operations per second respectively in our experiments. With the capacity being set, NVSwap can automatically determine whether to admit a process given the existing total I/O demand $DemandIOPS$ and the latency bound of the incoming process.

Reducing writes to NVMM: Many existing approaches have been proposed to reduce the number of writes to NVMM during paging, thus increasing its lifetime [12], [13]. In NVSwap, we focus on the software design related to the enforcement of paging latency. However, we believe our idea can also benefit from those schemes. For example, without violating the latency requirement, NVSwap may swap out less-frequently accessed pages or read-only pages (e.g., those store program codes) to persistent write buffers.

VI. Evaluation

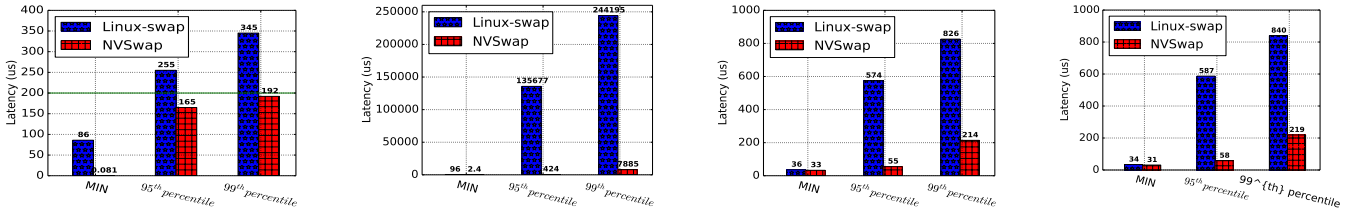
A. System Setup

We implemented NVSwap in the Linux kernel-4.4.241, which is a long-term state version. We instrumented the Linux /proc file system to pass the value of X^{th} percentile latency bound specified by users for the corresponding processes to the kernel. By default, processes are not latency-sensitive. Other code modifications are in the virtual memory management system, for example in the do_swap_page() function for in-situ paging, and in the shrink_page_list() function to select reclaimed pages using the latency control module.

For the experiments, we used a server that is configured with one 8-core Intel Xeon Scalable Silver 4208 2.1 GHz CPU, 32 GB DRAM, two Intel Optane DC Persistent Memory 128 GB modules, one 240 GB SSD (Samsung PM883 Series SATA 6 GB/s), and one 1 TB hard disk (Seagate Barracuda 7200.12). The hard disk is used to host the operating system. The SSD is used to host the regular-zone. In most of the experiments, we configured the computer so that the kernel can only address 5 GB DRAM as the main memory. A reserved NVMM space of 10 GB is used to host the persistent write buffer and the NV-zone.

We used the *YCSB* [7] benchmark for benchmarking in our experiments. YCSB is a framework developed for benchmarking cloud system performance. It provides a YCSB client for workload generation and a variety of database backends. In the experiments, we use $memcached$, an in-memory key-value database, as the default backend [26]. Unless otherwise specified, we use Workload A with 50/50 read/update ratio by default. The database record size is 1 KB. Its request distribution is zipfian. Both of the total $recordcount$ and $operationcount$ are set to 3 million. To demonstrate the practical effectiveness of NVSwap, we experimented with other Workloads including B, C, and F. We also varied the I/O characteristics of workloads, e.g., DB record size and operation types. We use both dedicated workloads and concurrent workloads to generate paging requests. Please find the configurations of the corresponding experiments in the following sections.

NVSwap is also effective with other applications (e.g., ImageMagick which is a software package providing command-line functionality for image editing). Due to the space limitation, we did not show their results here.



(a) OS Page In (b) OS Page Out (c) DB Update Records (d) DB Read Records

Fig. 4. (a), (b), (c), and (d) show the minimum, 95^{th} percentile, and 99^{th} percentile latency of OS page-in, page-out, DB update, and DB read records respectively with one instance of memcached. We set the 99^{th} percentile latency bound of page-in requests to be 200 us which is indicated by the green line.

B. Latency Enforcement

In this section, we present several experiments based on the YCSB benchmark that show the effectiveness of NVSwap in enforcing the latency bounds with both single and concurrent workloads.

1) Single Workloads: We study the effectiveness of latency control using NVSwap with a single memcached server accessing the swap space in the experiment. We compare NVSwap to the Linux swap system without latency control. For NVSwap, we set the 99^{th} percentile latency bound to be 200 us for page-in requests. Figure 4 shows the results. We have the following observations. First, from Figure 4(a), we observe that the 99^{th} percentile latency is reduced from 345 us to 192 us, which is below the latency bound 200 us. The result shows the effectiveness of NVSwap in enforcing latency bounds. It achieves the QoS goal by synergistically managing the disk scheduling queue and NVMM allocation. For example, the memcached server wrote 418,577 pages to the persistent paging buffer. The minimum latency of page-in requests is reduced from 86 us to 0.08 us because serving the requests using NVMM has a smaller latency than using SSDs. Second, from Figure 4(b), we observe that the 95^{th} and 99^{th} percentile latency of page-out requests are reduced by 99.6% and 96.7% respectively. It shows that using the persistent paging buffer can significantly alleviate the write interference on SSDs and reduce the congestion in the disk scheduling queue while also exploring the locality of workloads. Third, the latency of application-level requests was reduced by up to 74%. And the 99^{th} percentile latency of DB read and update is comparable to that of OS page-in requests.

Workload	*Read%*	*Update%*	*99% Latency Bound*
A	50%	50%	1000 us
B	95%	5%	500 us
C	100%	0%	300 us

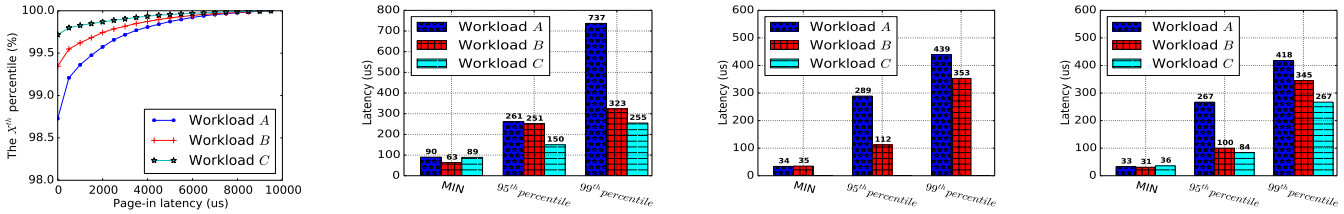
TABLE I
CONFIGURATIONS OF WORKLOAD A, B, AND C.

2) Concurrent Heterogeneous Workloads: In this section, we study the effectiveness of NVSwap using concurrent heterogeneous workloads. In the experiment we run three YCSB workloads A, B, and C. We show their configurations in Table I. The DB record size is 1 KB. The results are shown in Figure 5. We observed that the 99^{th} percentile latency of page-in requests is 737 us, 323 us, and 255 us for Workload A, B, and C respectively. They are smaller than their corresponding latency bounds, indicating that the QoS requirements were met with the help of NVSwap. In addition, because of the larger ratio of DB read operations in the workloads leading to a higher cache hit ratio, the 99^{th} percentile latency of DB operations is on average 33% smaller than the respective latency bound.

C. Changing Latency Bounds Dynamically

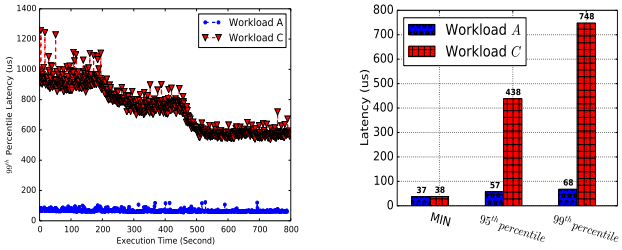
In this experiment, we show how the latency bounds set dynamically at the process level are respected. For this experiment, we ran two Workloads A and C sharing the swap space. Their initial 99^{th} percentile latency bounds are set to 100 us and 1000 us for Workload A and C respectively. Then the latency bound of Workload C is changed from 1000 us to 800 us at t = 200 second and from 800 us to 600 us at t = 450 second. Figure 6(a) shows the tail latency of DB operations in each 1-second time window during the execution of the two workloads.

At the start, the tail latency of the workloads matches the initial latency bounds as expected. Because of the latency bounds, the persistent paging buffer was used to serve Workload A and the regular-zone was used to serve Workload C. After the latency bound was reduced from 1000 us to 600 us for Workload C, more of its pages were directed to the persistent paging buffer to meet its QoS requirements. For example, after the tail latency became 600 us at t = 450 second, we see up to 90% of paging requests were served by the persistent paging buffer. The latency of the other workload Workload A was not affected showing the strong performance isolation between the two workloads. The overall paging performance is shown in Figure 6(b). We also observe that the measured 99^{th} percentile latency of Workload A is smaller than 100 us. For Workload C, its 99^{th} percentile latency is 748 us overall. This experiment shows the latency bound can be dynamically set and enforced by NVSwap during the execution of processes. Performance isolation is achieved between latency-sensitive paging processes.



(a) CDF (b) OS Page In (c) DB Update Records (d) DB Read Records

Fig. 5. (a) shows the latency distribution with three instances of memcached workloads. The 99^{th} percentile latency bounds are set to 1000 us, 500 us, and 300 us for workload A, B, and C respectively. (b), (c), and (d) show the minimum, 95^{th} percentile, and 99^{th} percentile latency of OS page-in, DB update, and DB read records. Workload C does not have DB update operations.



(a) (b)

Fig. 6. The 99^{th} percentile latency bound of Workload C is changed from 1000 us to 800 us at t = 200 second and is changed from 800 us to 600 us at t = 450 second. (a) shows the measured tail latency of DB operations in each 1-second time window. (b) shows the overall performance.

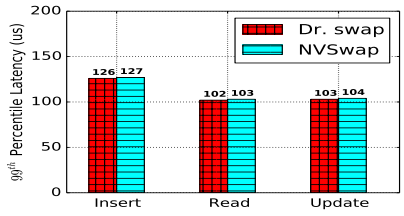


Fig. 7. Comparison of latency by NVSwap versus Dr. swap.

D. Comparison with Other Systems

We compared NVSwap with other state-of-the-art systems that support swapping using NVMM. Among them, we choose to implement Dr. swap as it is a page-level solution and provides direct read from NVMM, which is similar to the in-situ paging used in NVSwap. Because Dr. swap was not designed to provide latency enforcement, we only study the performance of NVSwap without using the latency control module. In the experiment, both Dr. swap and NVSwap only access NVMM for paging. No regular-zone is configured. We ran two instances of Workload A concurrently accessing NVMM. The 99^{th} percentile latency of DB operations is shown in Figure 7. Since the kernel-level tracing is disabled, we did not show the latency of page-in requests for the fairness of the study. The results show that the tail latency of NVSwap is 0.8% higher than that of Dr. swap. The reason is that NVSwap needs to copy the page from the persistent write buffer to the NV-zone, which is then mapped to process address spaces. In contrast, Dr. swap directly mapped it without the additional latency of memory copy.

VII. Conclusion

In this paper, we studied the problem of latency-aware paging in the virtual memory of operating systems. We propose a novel paging scheme called NVSwap which provides a cost-effective and hybrid swap space using both NVMM and SSD. It allows the setting of X^{th} percentile page-in latency bound for a single process or a group of processes. NVSwap controls the host swap device, the memory page reclaim rate, the scheduling queue size in DRAM, and the allocation of persistent paging buffer in NVMM for paging processes. We implemented NVSwap in Linux kernel-4.4.241. Our evaluation with a diverse set of YCSB workloads shows that NVSwap can enforce the tail latency while providing strong performance isolation for latency-sensitive processes.

Acknowledgment

This research was supported by US National Science Foundation under CNS 1906541 and DoE Electricity Industry Technology and Practices Innovation Challenge. This work was also funded in part by WSU Vancouver Research Grant.

References

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 9th ed. Wiley Publishing, 2012.
[2] M. Saxena and M. M. Swift, "Flashvm: Virtual memory management on flash," in *ATC'10*.
[3] D. X. Memory, https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html, 2019.
[4] "Intel optane dimms," https://blocksandfiles.com/2018/12/13/intel-confirms-optane-dimm-and-ssd-speed/a, 2018.
[5] D. Bovet and M. Cesati, *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.
[6] "Tunable watermark," https://lwn.net/Articles/422291/ , 2011.
[7] "Yahoo! cloud serving benchmark," https://github.com/brianfrankcooper/YCSB/wiki , 2018.
[8] S. Park and K. Shen, "Fios: A fair, efficient flash i/o scheduler," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12.
[9] "Intel announces cascade lake: Up to 56 cores and optane persistent memory dimms," https://www.tomshardware.com/reviews/intel-cascade-lake-xeon-optane,6061-3.html , 2019.
[10] J.-Y. Jung and S. Cho, "Memorage: Emerging persistent ram based malleable main memory and storage architecture," in *ICS'13*.

[11] K. Zhong, X. Zhu, T. Wang, D. Zhang, X. Luo, D. Liu, W. Liu, and E. H. . Sha, "Dr. swap: Energy-efficient paging for smartphones," in *ISLPED'14*, 2014.
[12] X. Chen, E. H. . Sha, W. Jiang, Q. Zhuge, Junxi Chen, Jiejie Qin, and Yuansong Zeng, "The design of an efficient swap mechanism for hybrid dram-nvm systems," in *EMSOFT'16*.
[13] K. Zhong, D. Liu, L. Long, J. Ren, Y. Li, and E. H. Sha, "Building nvram-aware swapping through code migration in mobile devices," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 11, 2017.
[14] X. Zhu, D. Liu, K. Zhong, Jinting Ren, and T. Li, "Smartswap: High-performance and user experience friendly swapping in mobile systems," in *DAC'17*.
[15] W. Guo, K. Chen, H. Feng, Y. Wu, R. Zhang, and W. Zheng, "Mars: Mobile application relaunching speed-up through flash-aware page swapping," *IEEE Trans. Comput.*
[16] G. Zhu, K. Lu, X. Wang, Y. Zhang, P. Zhang, and S. Mittal, "Swapx: An nvm-based hierarchical swapping framework," *IEEE Access*, vol. 5, pp. 16 383–16 392, 2017.
[17] A. Awad, S. Blagodurov, and Y. Solihin, "Write-aware management of nvm-based memory extensions," in *ICS'16*.
[18] J. Zhang, A. Riska, A. Sivasubramaniam, Q. Wang, and E. Riedel, "Storage performance virtualization via throughput and latency control," in *MASCOT'05*.
[19] N. Li, H. Jiang, D. Feng, and Z. Shi, "Pslo: Enforcing the xth percentile latency and throughput slos for consolidated vm storage," in *EuroSys'16*.
[20] X. Zhang, K. Davis, and S. Jiang, "Qos support for end users of i/o-intensive applications using shared storage systems," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.
[21] S. Karki, B. Nguyen, and X. Zhang, "Qos support for scientific workflows using software-defined storage resource enclaves," in *IPDPS'18*.
[22] A. Gulati, G. Shanmuganathan, X. Zhang, and P. Varman, "Demand based hierarchical qos using storage resource pools," in *ATC'12*.
[23] A. Gulati, I. Ahmad, and C. A. Waldspurger, "PARDA: Proportional allocation of resources for distributed storage access," in *FAST'09*.
[24] M. Gorman, *Understanding the Linux Virtual Memory Manager*, 2004.
[25] "The iometer benchmark," http://www.iometer.org , 1998.
[26] "A high-performance, distributed memory object caching system," https://memcached.org/ , 2019.