Userland CO-PAGER: Boosting Data-Intensive Applications with Non-volatile Memory, Userspace paging

Feng Li
Purdue University
Indianapolis, Indiana
li2251@purdue.edu

Daniel G. Waddington
IBM Research, Almaden
San Jose, California
daniel.waddington@ibm.com

Fengguang Song
Indiana University-Purdue University
Indianapolis, Indiana
fgsong@iupui.edu

ABSTRACT

With the emergence of low-latency non-volatile memory (NVM) storage, the software overhead, incurred by the operating system, becomes more prominent. The Linux (monolithic) kernel, incorporates a complex I/O subsystem design, using redundant memory copies and expensive user/kernel context switches to perform I/O. Memory-mapped I/O, which internally uses demand paging, has recently become popular when paired with low-latency storage. It improves I/O performance by mapping the data DMA transfers directly to userspace memory and removing the additional data copy between user/kernel space. However, for data-intensive applications, when there is insufficient physical memory, frequent page faults can still trigger expensive mode switches and I/O operations. To tackle this problem, we propose CO-PAGER, which is a lightweight userspace memory service. CO-PAGER consists of a minimal kernel module and a userspace component. The userspace component handles (redirected) page faults, performs memory management and I/O operations and accesses NVM storage directly. The kernel module is used to update memory mapping between user and kernel space. In this way, CO-PAGER can bypass the deep kernel I/O stacks and provide a flexible/customizable and efficient memory paging service in userspace. We provide a general programming interface to use the CO-PAGER service. In our experiments, we also demonstrate how the CO-PAGER approach can be applied to a MapReduce framework and improves performance for data-intensive applications.

CCS CONCEPTS

• **Software and its engineering → Memory management**;

KEYWORDS

Non-volatile Memory, Memory Management, Big Data

1 INTRODUCTION

A new generation of non-volatile memory (NVM) technology is driving the advent of storage devices with unprecedented throughput and latency characteristics. For example, the Intel Optane DC P4800X NVMe SSD [1], attaching directly to the PCIe bus, has a 4KiB block random read/write access latency (queue depth 1) of $7-10\mu sec$ and throughput of up to 550 KIOPS. However, the legacy storage subsystems in current operating systems are unable to unlock the full potential of these devices due to incurred software overhead [2].

Eliminating traditional IO stack overhead is an active research topic[3, 4]. Among the proposed methods, memory-mapped I/O has become a popular alternative to traditional file I/O. By making use of the demand paging mechanism of the Linux kernel, memory-mapped I/O can eliminate the extra data copy and reduce context switches. The memory-mapped I/O approach also provides a straightforward path to consumption but still incurs software overhead from the kernel-based page mapping process [5, 6].

The challenge we address in this work is how to improve memory paging performance by leveraging the low-latency benefits of these new devices. Instead of improving kernel paging mechanism itself, we propose to migrate the paging functionality into userspace. This not only allows us to optimize the IO path, but also customize the paging function for a specific application (e.g., application-specific pre-fetching).

Our CO-PAGER solution consists of a paging component that executes in userspace, and a minimal kernel module. The userspace component detects page faults, manages and performs paging (between main memory and low-latency storage). The kernel component updates the memory-mapping information (page tables).

This design brings several benefits:

- Since I/O now "bypasses" the kernel, we avoid the overhead introduced by the conventional I/O design.
- Userspace now has more control over paging mechanisms. For example, given a domain-specific application which has specific memory access pattern, a customized prefetch and/or replacement policy can be easily implemented in userspace without costly modification to the kernel.

To demonstrate the benefits of the CO-PAGER design and show how it can be used with existing applications, we have prototyped NV-Phoenix, which is an integration of CO-PAGER with the Phoenix MapReduce framework [7]. More details of NV-Phoenix will be given in Section 4.2 and 5.2.

Note that the usage of CO-PAGER is not limited to MapReduce. In NV-Phoenix, we introduce a portable interface, which can be reused with other existing projects written in general languages like C/C++. This way, application developers don't need to know details about the NVM device under the hood. Instead, they can take advantage of fast storage devices to accelerate their applications, by using the familiar interface such as malloc and free.

HP3C '19, March 8–10, 2019, Xi'an, China

ACM ISBN 978-1-4503-6638-0/19/03…$15.00
https://doi.org/10.1145/3318265.3318272

To the best of our knowledge, this work makes the following contributions:

- CO-PAGER: a lightweight userspace memory service, that leverages user-level IO with high-performance NVMe devices to perform fast paging.
- NV-Phoenix: an improved MapReduce framework, which is integrated with CO-PAGER service, and enables existing MapReduce applications to run without API changes.
- A general persistent programming interface, that provides simplified access to low-latency NVM SSDs.

In the remainder of the paper, the following section introduces the background of NVMe storage technologies, and knowledge of paging and virtual memory. Section 3 introduces related work and compare them with ours. Section 4 shows the design of CO-PAGER memory service and how we integrate it with the Phoenix MapReduce framework. Finally, Sections 5 and 6 present the experimental results and summarize the paper.

2 BACKGROUND

In this section, we first introduce the background and current usage/applications of NVMe SSDs, followed by some fundamental knowledge of virtual memory and paging and their applications.

2.1 NVMe SSDs and userspace device driver

NVMe (NVM express, [8]) is an optimized, high-performance host controller interface for PCIe-based SSDs. It's designed to provide efficient access to next-generation NVM storage devices, and it addresses several performance vectors: bandwidth, IOPS and latency, scalability, etc. Unlike traditional protocols such as SAS and SATA, NVMe protocol features multiple deep queues and can provide high IOPS per CPU instruction and lower I/O latency [9]. It supports both enterprise(e.g. reservation) and client features (e.g. power management)

Native driver support has been available in both Windows and Linux (since 3.3, Jan 2012). To better accommodate the fast NVMe devices, the new blk-mq framework has been added to Linux kernel since 3.13, which will bypass previous Linux I/O scheduler and provide functions to device drivers for mapping I/O inquiries to multiple queues.

IBM Comanche [10, 11] is a framework for user-level compositional storage development. It provides an approach to reorganizing different build-blocks (e.g. block allocation, caching, partitioning) to provide custom storage solutions for various needs from applications.

We use the blk-nvme component from the Comanche framework to interact with NVMe SSDs in userspace. Internally, Comanche blk-nvme component uses tools such as Intel SPDK [12] and DPDK [13] to manage memory and I/O operations in userspace.

2.2 Paging and virtual memory

Virtual memory provides an address mapping from virtual address space to physical address space. It offers protection of memory regions in two levels: memory mappings of different processes are not overlapped; kernel memory is isolated with userspace processes. Paging is an important part of virtual memory implementation, and by swapping from/to the secondary storages like hard drives, it provides userspace processes an "illusion" of available memory, which can be larger than physical memory available in the system. However, such paging procedure with fast storage devices like NVMe SSDs can be challenging, due to the legacy kernel design which was originally for slower spinning hard drives. User-level paging implementations usually offer flexibility and speed. Examples of user-level paging system are DLM [14], micro-kernel kernel implementations [15, 16], etc.

A userspace application can request virtual memory from operating system kernel, through system calls such as mmap or sbrk/brk. However, because of the overhead of context switch during system calls, it's better to allocate a large chunk of memory and then split it to get small chunks. Many library calls, such as C malloc/free, can help us with that, by maintaining lists of pre-allocated buffers. Besides malloc provided by stand C library, there are also some other available allocators which also provide equivalent POSIX-compatible malloc/free interfaces, which applications can use interchangeably. By providing the same malloc/free interfaces, our CO-PAGER service can also be applied to existing applications easily.

2.3 MapReduce and Phoenix

MapReduce is a general programming model designed for parallelized computation [17]. The basic idea is to take a set of key/value pairs as input, "map" them to intermediate key/value pairs, and then merge the intermediate values based on the keys. The programming model can fit well to many parallel problems and it has influenced essential tools in big data domain, such as Apache Spark [18]and Hadoop [19]. MapReduce usually utilizes a cluster of nodes and communications between nodes are established using remote procedure calls or distributed file system, both of which can be expensive for data-intensive applications.

To eliminate such expensive communication between nodes, one approach is to align tasks to a single node. With multi-core and multi-processing systems (e.g. Intel Xeon Phi) becoming even more popular recently, the computation power of a single node has dramatically increased. And it's promising to run data-intensive applications in a single node, using MapReduce framework like Phoenix [7]. Unlike the cluster-based MapReduce frameworks, Phoenix uses threads to spawn parallel map and reduce tasks.

Since all the threads are spawn in a single node, main memory can become the performance bottleneck [20, 21]. If the active memory of an application cannot fit into the main memory, in-memory pages can be swapped out to secondary storage periodically. Historically, swapping is considered relatively slow since legacy drives such as spinning drives usually have limited bandwidth and latency. Now that the low-latency NVM storage is available, we can use it to accelerate the Phoenix framework, with the help of Linux paging mechanism.

3 RELATED WORK

There have been several explorations on using NVM device to construct a "persistent heap". Hwang et al. [22] designed a persistent object store, along with a namespace and persistent object management scheme, and auto-generation is supported for easier use of the object store. Coburn et al [4] proposed *NV-Heaps*, which is a

lightweight and high-performance persistent object system, and also ensures safety in the same time. *Megalloc* [23] is a distributed NVM allocator, which can expose virtual address space of NVMs installed in multiple nodes, by using RDMA (remote direct memory access). User-level memory page allocator [24], shows how user-level virtualized MMU can be used to outperform state-of-art memory allocators. Markthub et al. designed *Dragon* [25], which transparently extends capabilities of GPU memory by mapping NVM storage to GPU addressable space. Unlike those work, our CO-PAGER focuses on the kernel-bypass design and utilizes more simplified I/O stacks in userspace.

The persistent memory programming [26] provides a growing collection of libraries (Persistent Memory Development Kit, PMPK [27]), which can be exposed to applications so that they can manage data among volatile memory, persistent memory and storage explicitly. Moon et al. [28] explore how to optimize a Hadoop MapReduce Framework with SSDs, by utilizing fast SSDs for intermediate Hadoop data and slower HDDs for Hadoop File System(HDFS).

Micro-kernel [29] provides an alternative way to support I/O in userspace. Unlike monolithic kernels (e.g. Linux), micro-kernels keep most of the system services in user space, and communications happen in the form of message passing. Micro-kernels usually suffer from poor performance due to the expensive inter-process communications.

4 METHODS

In this section, we first introduce a top-level system design of CO-PAGER memory service, followed by more details of implementation. After that, we introduce the new system of NV-Phoenix and the general programming interface of the CO-PAGER service.

4.1 System Design of CO-PAGER

The fundamental idea of CO-PAGER is to lift the paging service from kernel to userspace as much as possible, which is demonstrated in Figure 1. The left part of the figure shows a brief design for current Linux kernel paging mechanism, as we have seen in subsection 2.2. The Linux kernel manages several critical components: page replacement and I/O path. Page tables are maintained for each process and they are used by the kernel to track whether a page is mapped to physical memory or swapped out to the disk. The block device (i.e. NVMe drive in our work) is totally managed by Linux kernel through a kernel driver (blk-mq).

We present our new design in the right part of Figure 1, where we lift the paging service to userspace, and now the paging service can interact with NVMe SSDs in userspace, through the Comanche blk-nvme interface (shown as "Device Driver" in the userspace).

Figure 2 shows more details of the CO-PAGER design, where the readers can see more details of each component involved in the CO-PAGER service.

4.1.1 Kernelspace components. We introduce the kernelspace component first, because it's relatively simple and will be used by the userspace service. We create a kernel module called XMS (eXtreme Memory Service), whenever the userspace component decides to map a virtual address range to physical memory frames, it sends the corresponding virtual and physical address information to XMS module, and then XMS updates the in-kernel mapping

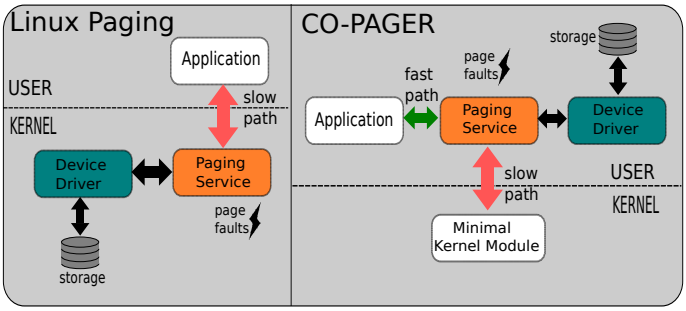

Figure 1: Comparison between kernel paging and userland CO-PAGER.

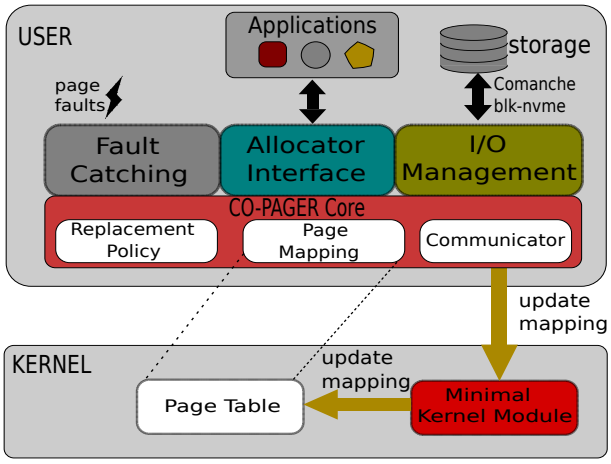

Figure 2: Implementation details of CO-PAGER.

information (page table, TLB, etc) correspondingly. The in-kernel memory mapping information is still required so that when the application accesses the mapped pages for the second time, it won't trigger faults again.

4.1.2 Userspace Components. There are three major tasks for the userspace components: 1. capture faults 2. handle faults 3. synchronize mapping information with the kernel.

Page faults are captured by *fault catching* component, which is actually a segmentation fault signal handler. When an application allocates memory space from CO-PAGER service, CO-PAGER first asks kernel for an empty range of virtual addresses, and sets the protection bit of this range as "PROT_NONE". This means initially the application doesn't have READ/WRITE access for the allocated addresses and any further accesses will trigger segmentation faults. Such segmentation faults later will be captured by the *fault catching* component, and then be solved by the *CO-PAGER core*.

The *CO-PAGER core* component is where page faults are actually handled. *CO-PAGER core* reserves physical memory frames and has replacement policy and page mapping mechanisms to map a virtual address either to (1) a physical memory frame or (2) a NVMe logical block address (similar to kernel paging, as we can see from Subsection 2.2). To solve a page fault, *CO-PAGER core* first selects a victim page, then updates mappings and issues I/O through the *I/O*

management path. The *I/O management* path uses the Comanche blk-nvme interface we have mentioned in Subsection 2.1. which issues asynchronous I/O requests and directly interacts with the NVMe device in userspace.

CO-PAGER also needs to synchronize the mapping information to the kernel (the XMS module we have mentioned). This is done by the *communicator*, which keeps the file descriptor of the XMS module, and exchanges memory mapping information with the XMS module.

The allocator interface above the *CO-PAGER core* provides applications a generic API like malloc/free. This is the only interface the applications need to interact with, and it will be introduced later.

4.2 NV-Phoenix and general memory allocator interface

In this subsection, we describe how we design NV-Phoenix, a MapReduce framework with CO-PAGER enabled, so that memory allocation can be managed easily with CO-PAGER memory service mentioned above.

To better fit our CO-PAGER into this ecosystem, we compose the CO-PAGER userspace component as a shared library, which provides POSIX-compatible malloc/free interfaces, so that the applications don't need any explicit code modifications other than linking to the CO-PAGER library. This is achieved by adding a 'mmap replacement' to TCMalloc [30], using the '*MallocHook:: SetMmapReplacement*' function. The default behavior of TCMalloc is asking memory from kernel using *mmap* or *sbrk* system calls. There are several reasons why we choose TCMalloc instead of Glibc malloc. Firstly, TCMalloc has clearer code structure and it's easier to add a mmap replacement; also, there are more flexible controls provided by TCMalloc. For example, setting the *TCMALLOC_SKIP_SBRK* environment variable can force TCMalloc to ignore sbrk/brk system calls, and obtain memory from kernel only through mmap system call. After we set the *TCMALLOC_SKIP_SBRK* environment variable and add our CO-PAGER mmap hook, all memory allocation related system calls from TCMalloc will be translated to CO-PAGER allocations.

In our earlier implementation, we tried to intercept "malloc" as CO-PAGER allocation directly, using the LD_PRELOAD trick [31]. However, doing that we will lose all the optimizations provided by userspace memory allocation libraries like Glibc malloc or TCMalloc.

Performance comparison of mmap-based TCmalloc and CO-PAGER-based Tcmalloc are presented in Section 5, by linking both libraries with the Phoenix MapReduce framework.

5 EXPERIMENTS

To show the advantage of the userspace paging design, we first use a micro-benchmark to compare the page-fault-handling latency of CO-PAGER to the default Linux paging subsystem. Then a real-word application, using the NV-Phoenix introduced in section 4.2, is tested with various input sizes. We also compare the end-to-end time of the NV-Phoenix application with the same application running with other memory conditions.

System Configuration Table 1 shows the system configurations used in the experiments. We use Intel Optane 900P NVMe SSD [32], which was launched in 2017. It has symmetric read/write latency of 10 us, and can deliver 550*K* IOPS. [1]

Table 1: Machine Configuration

Type	Information
CPU	Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz quad-core processors
Main memory	8GB
OS version	Ubuntu 16.04.1
Kernel version	4.13.0-38
Compiler	GCC 5.4.0
Secondary storage #1	WD SATA3 5400 rpm
Secondary storage #2	Intel 900P 280GB AIC

5.1 Micro-benchmark

The first experiment is to measure the time used by CO-PAGER to solve each page fault, and compare with the traditional Linux paging utilities (more specifically, the *mmap* system call). The micro-benchmark tool we use here can access a contiguous memory region in different modes. Sequential and random modes write to pages in sequential and random order, while the stride mode writes with a stride size of 8 pages.

Next we explain how we create the virtual memory region for the benchmark. For the CO-PAGER group, we use the "raw" CO-PAGER allocation interface to create a heap, without using TCMalloc's *mmap* replacement. For the mmap group, we use the *mmap* system call with the MADVISE_RANDOM option [2], We don't use memory allocator interfaces (such as TCMalloc malloc/free) here because those library calls maintain internal buffers and other optimization techniques, which make tracking fault-handling more difficult.

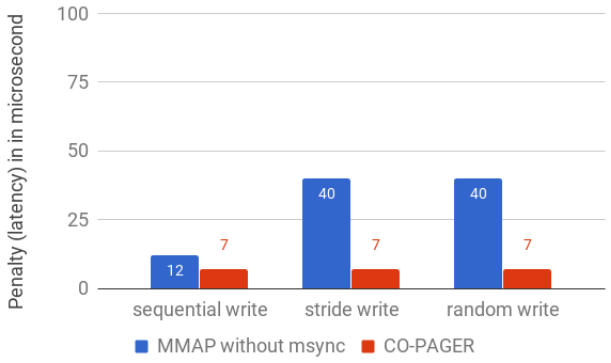

Figure 3: Write fault penalty for cached access.

Figure 3 shows the latency for three types of writes. From this figure, CO-PAGER has reduced the fault penalty by 41.6%, 82.5% and 82.5% for random/sequential/stride write respectively. Since

[1] unlike enterprise version of Optane SSDs, 900P can only be configured with 512B logical block size

[2] mmap's performance might be better with other MADVISE or MMAP_POPULATE options, due to more optimizations in kernel.

Linux paging utilizes the page cache, the paging operation does not have to update its contents into the secondary device (NVMe SSD) immediately. Instead, updates will be resident in page cache, and be flushed out in a later time. We simulate this behavior in CO-PAGER: instead of flushing into NVMe device each time for a paging out, we only issue one asynchronous write command and return immediately. The completion of that write will be checked the next time when the same physical page is used.

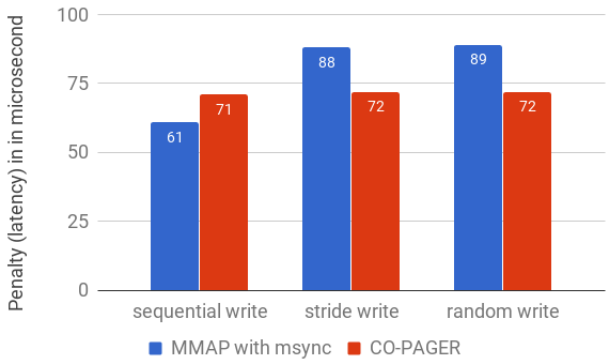

Figure 4: Write fault penalty for sync access.

Even though Figure 3 shows how CO-PAGER reduces the kernel (software) overhead, the latency shown is not directly related to the NVMe devices, since all the writes are "asynchronous". However, real applications do more than just writing and we also need consider effects of actual flushing. With this in mind, we add synchronization for both CO-PAGER and Linux kernel paging. So that after each time they both solve a fault, the update is guaranted written into the NVMe device. In Linux kernel paging, this is enforced by adding a "msync" after updating each page. The new results are shown in Figure 4, From the figure, we can observe that, though CO-PAGER is faster than Linux kernel paging in both stride writes and random writes (19% less fault handling latency), it is slightly slower in the case of a sequential write. The reason might be the kernel optimizations of I/O operations, such as the merging I/O requests for adjacent sectors in I/O scheduler [33]. Currently CO-PAGER is in its prototype status and more optimizations can be made in future. Note that since the 900P has 512 logical block size, we expect an even lower latency for CO-PAGER if using enterprise NVMe device such as Intel Optane DC P4800X [1].

5.2 Evaluation of NV-Phoenix

There are totally 7 applications provided in Phoenix package, all of them can be configured with CO-PAGER. We show the performance of *k-means* application as an example in this subsection.

In figure 5, we run K-means (from the recent C++ version of Phoenix package [21]) with different element sizes, under different memory constraints:

(1) mmap with sufficient memory
(2) mmap with 32M physcial memory, paging on NVMe SSD using Linux default paging mechanism.

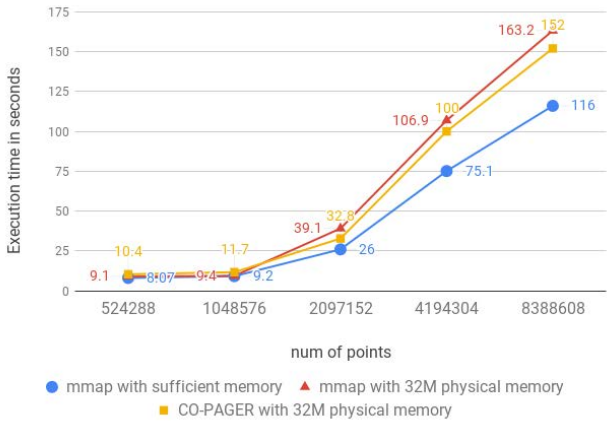

Figure 5: K-means execution time with Phoenix.

(3) CO-PAGER with 32M physical memory, paging on NVMe SSDs using the proposed CO-PAGER userspace paging design.

Here both mmap and CO-PAGER are used together with TCMalloc library (different from the "raw" performance we saw from the first experiment). It means that instead of triggering a *mmap* system call or CO-PAGER allocation for each malloc/new operation, the TCMalloc library accumulates the small allocations and allocates a large chunk of memory once.

The memory constraints are enforced using the *cgroup* tool [34]. K-means application is configured with 100 clusters and each point has three dimensions. The results of running Phoenix K-means are shown in Figure 5, with input size from 512*K* points to 8192K points. From the figure, we can see that the running time under various memory constraints is the same when the number of points is smaller than 2M. That's reasonable since the active memory of the application can fit in the 32MB physical memory.

However, when input size keeps increasing, we can see the significant difference of various memory constraints. *Mmap* with sufficient memory still gives the shortest execution time, which is expected since the active memory can always fit in the physical memory. For mmap with 32M memory constraint using kernel paging, the k-means application runs slower, because it needs to page in/out from/to the NVMe device. Under the same 32M memory constraint, CO-PAGER allocator's performance is better than Linux mmap, thanks to the simpler design of paging service in userspace.

6 CONCLUSION

The low-latency and high-throughput of NVMe devices have made us to rethink of I/O stack design of modern operating systems. Directly mmap to fast NVMe storage would produce significant software overhead, since the paging system was originally designed for slower hard drives. To better fit NVMe SSDs using the extended memory model, we introduce CO-PAGER, which is a light-weight memory service and consists of an efficient user-level memory service and a minimal kernel module. We lift the paging and I/O management to the user space and use the minimal kernel module

to update the in-kernel page table. This design provides users with higher flexibility to use more customized memory service based on the different memory access patterns of various applications. The experiments show that the CO-PAGER significantly reduces the page fault penalty, compared with the paging utilities in a recent kernel. The CO-PAGER is also integrated with Phoenix, a shared-memory MapReduce framework, to demonstrate that the general APIs can be easily applied to other real applications. Our results show the CO-PAGER memory service can significantly improve the performance of the application under high memory pressure, compared to the default kernel paging service.

7 FUTURE WORK

When we are comparing the Linux memory map I/O with raw CO-PAGER allocation, we found there are many tunable options which can affect the performance of memory-mapped I/O. One such example is the *madvise* system call, which can tell the kernel the desired access pattern of the application. To our surprise, those hints don't always bring possible effects in our preliminary experiments, when there is not enough physical memory to accommodate all the active pages. Currently we are investigating more detailed patterns of the memory-mapped I/O behavior, and we think it will help us better improve the CO-PAGER design.

In the experiment section, we are using the simplest direct mapping when we manage the replacement of pages. Since the pager itself is modularized, it is very intuitive to implement other types of replacement policies for various types of workloads. And we expect we can get better results for better-tuned replacement policies.

ACKNOWLEDGMENTS

This material is based upon research partially supported by Purdue Research Foundation and by the NSF Grant #1835817. We would like to thank the Storage Systems group in IBM Research Almaden, for their assistance during the development of CO-PAGER.

REFERENCES

[1] Intel SSD DC P4800X specifications. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series/p4800x-375gb-aic-20nm.html, 2018.
[2] Daniel Waddington and Jim Harris. Software challenges for the changing storage landscape. *Communications of the ACM*, 61(11):136–145, 2018.
[3] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 91–104. ACM, 2011.
[4] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices*, 46(3): 105–118, 2011.
[5] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient memory mapped file i/o for in-memory file systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association, 2017.
[6] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. Efficient memory-mapped i/o on fast storage device. *ACM Transactions on Storage (TOS)*, 12(4):19, 2016.
[7] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24. Ieee, 2007.
[8] Danny Cobb and Amber Huffman. NVM Express and the PCI Express SSD Revolution. In *Intel Developer Forum*. Intel, 2012.
[9] NVM Express Overview. http://nvmexpress.org/wp-content/uploads/NVMe_Overview.pdf, 2018.
[10] Daniel G Waddington. Fast & flexible io: A compositional approach to storage construction for high-performance devices. *arXiv preprint arXiv:1807.09696*, 2018.
[11] IBM Comanche project. https://github.com/IBM/comanche, 2018.
[12] Storage performance development kit (SPDK). http://www.spdk.io, 2018.
[13] Data Plane Development Kit Project. https://dpdk.org, 2018.
[14] Hiroko Midorikawa, Yuichiro Suzuki, and Masatoshi Iwaida. User-level remote memory paging for multithreaded applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 196–197. IEEE, 2013.
[15] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9): 70–77, 1996.
[16] Dylan McNamee and Katherine Armstrong. Extending the mach external pager interface to accomodate user-level page replacement policies. In *USENIX MACH Symposium*, pages 17–30, 1990.
[17] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[18] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.
[19] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.
[20] Richard M Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 198–207. IEEE, 2009.
[21] Justin Talbot, Richard M Yoo, and Christos Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*, pages 9–16. ACM, 2011.
[22] Taeho Hwang, Dokeun Lee, Yeonjin Noh, and Youjip Won. Designing persistent heap for byte addressable nvram. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2017 IEEE 6th*, pages 1–6. IEEE, 2017.
[23] Songping Yu, Nong Xiao, Mingzhu Deng, Yuxuan Xing, Fang Liu, and Wei Chen. Megalloc: Fast distributed memory allocator for nvm-based cluster. In *Networking, Architecture, and Storage (NAS), 2017 International Conference on*, pages 1–9. IEEE, 2017.
[24] Niall Douglas. User mode memory page management: An old idea applied anew to the memory wall problem. *arXiv preprint arXiv:1105.1815*, 2011.
[25] Pak Markthub, Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Satoshi Matsuoka. Dragon: breaking gpu memory capacity limits with direct nvm access. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 32. IEEE Press, 2018.
[26] Pmem.io-Persistent Memory Programming. https://pmem.io, 2018.
[27] Persistent Memory Development Kit (PMDK) project. https://github.com/pmem/pmdk/, 2018.
[28] Sangwhan Moon, Jaehwan Lee, Xiling Sun, and Yang-suk Kee. Optimizing the hadoop mapreduce framework with high-performance storage devices. *The Journal of Supercomputing*, 71(9):3525–3548, 2015.
[29] Jochen Liedtke. *On micro-kernel construction*, volume 29. ACM, 1995.
[30] TCMalloc : Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2018.
[31] Linux Programmer's Manual dynamic linker/loader. http://man7.org/linux/man-pages/man8/ld.so.8.html, 2018.
[32] Intel Optane SSD 900P Series. https://www.intel.com/content/www/us/en/solid-state-drives/optane-ssd-900p-brief.html, 2018.
[33] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.", 2005.
[34] Manual page for cgroup. http://man7.org/linux/man-pages/man7/cgroups.7.html, 2018.