NVMM cache design: Logging vs. Paging

Rémi Dulong*‡, Quentin Acher†, Baptiste Lepers*, Valerio Schiavoni*
, Pascal Felber*, Gaël Thomas‡
University of Neuchâtel, Switzerland* — ENS Rennes, France† — Télécom SudParis, France‡

***Abstract*—Modern NVMM is closing the gap between DRAM and persistent storage, both in terms of performance and features. Having both byte addressability and persistence on the same device gives NVMM an unprecedented set of features, leading to the following question: How should we design an NVMM-based caching system to fully exploit its potential? We build two caching mechanisms, NVPages and NVLog, based on two radically different design approaches. NVPages stores memory pages in NVMM, similar to the Linux page cache (LPC). NVLog uses NVMM to store a log of pending write operations to be submitted to the LPC, while it ensures reads with a small DRAM cache. Our study shows and quantifies advantages and flaws for both designs.**

I. Introduction

The emergence of modern NVMM is a great opportunity to implement known designs and adapt them, or invent new ones. We tried these two approaches with caching mechanisms for a file system stored in secondary storage. Indeed, caching data for slower tier storage devices (SSD or HDD) is a great use case for NVMM. It provides high persistence guarantees, higher read and write bandwidth and lower latencies than most persistent block devices [1]. In this study, we target applications that require a high level of data consistency, which would highly solicit a regular disk with frequent calls to `fsync`. For such applications, we propose a persistent cache able to give fast persistence guarantees without having to wait for a slow secondary storage.

II. NVMM-based Caching

NVPages and NVLog are POSIX-like IO shared libraries. They provide standard IO functions, such as `open`, `pread`, `pwrite`, `close`, *etc*. When the shared library is loaded, NVMM is mapped, and data structures are initialized. A flag in NVMM is set to 1 when the program is loaded, and set to 0 when it is unloaded properly. This flag allows both caches to start a recovery procedure in case of a previous crash, flushing to disk every modification still pending in cache when the crash occurred. So far, they do not support multiple threads. However, they differ in their core implementation, depicted in Fig. 1 and Fig. 2.

NVPages. NVPages is designed as a regular page cache, with a few adaptations to make it compliant with NVMM and its persistence guarantees. 4 KiB pages are stored in NVMM. When a page is accessed, a radix tree in volatile memory looks for a volatile metadata structure that contains a pointer to the non-volatile page. In order to ensure consistency after a crash, calls to `pwrite` first write data in a redo log stored in persistent memory. Then, the redo log content is flushed to

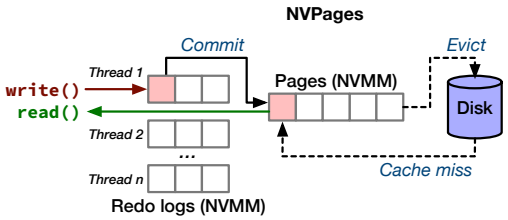

Fig. 1. Core design of NVPages

the actual non-volatile page cache. The page cache eviction is done with a least recently used (LRU) policy. NVPages can be used in `O_DIRECT` mode, bypassing the LPC to interact directly with the disk with aligned 4 KiB blocks. We do not report performance with this mode since we measured that bypassing the LPC reduces performance in read. As described in Fig.1, NVPages is designed to be adapted for multithreaded workloads.

NVLog. NVLog builds atop NVCache [2], ported to work as a shared library. It embeds two main components: a NVMM log, and a small DRAM page cache. When the `pwrite` function is called, data is written to the NVMM log. A background thread continuously waits for log entries and writes them to disk as soon as possible. To ensure consistency in this configuration, every call to `pread` should get the page from disk and check if patches (log entries) have to be applied before returning the data. As this would make reads very slow, NVLog keeps a small DRAM page cache (2 GiB) with up-to-date data. It also keeps track of pages that would need to be patched before returning, so it only searches in the NVMM log when necessary. For reads, NVLog uses the LPC as an extension of NVLog's DRAM cache, from which it can fetch data instead of waiting for the disk. For writes, NVLog submits changes to the LPC in batches, before calling fsync to ensure the data is persisted on disk. This way, it benefits from LPC optimizations, such as merging consecutive writes on the same offset before writing the page on disk. Its design is complex because of the internal synchronization between the application and the background thread. Adapting it for multithread remains challenging.

Discussion. NVLog is designed to absorb bursts of writes in its log, but may not be suited for mixed or parallel IOs. It only keeps a small amount of pages updated in DRAM. Increasing the amount of NVMM in NVLog does not change the probability of cache hit. Instead, NVPages is designed to maximize the probability of cache hit by keeping a lot of memory pages available in NVMM, as almost all of its

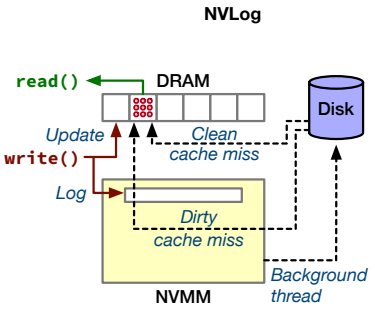

Fig. 2. Core design of NVLog

allocated NVMM is dedicated to pages. We expected the latter approach to be more efficient for mixed IOs, reducing the amount of interactions with the disk.

III. Evaluation

Our benchmark machine is a Supermicro mono-socket machine with an Intel Xeon Gold 6326 CPU, 2 modules of 128 GiB of Intel Optane v200 DCPMM [3], and a 512 GB NVMe SSD, running Ubuntu 20.04 LTS.

We evaluated our 2 systems with FIO [4]. These tests are performing 20 GiB of random accesses through a 20 GiB-wide file. In Fig. 3 and Fig. 4, we submit pure reads (`randr`), 50% reads and 50% writes (`randrw`), 90% reads and 10% writes (`randrw90`), and pure writes (`randw`). Then, to show the efficiency of the caching policies, we measure the same tests with a Zipfian distribution [5] that ensures 95% of random offsets will be in 5% of the file. Each bar is the average completion time of 5 runs. For each plot, we compare NVPages and NVLog with a given amount of NVMM allocated. Our reference is the regular `psync` IO engine of FIO which uses regular POSIX functions, measuring the performance of the LPC in DRAM. With this baseline, there is no guarantee of persistence, while NVPages and NVLog both guarantee persistence as soon as a `pwrite` call returns. Having similar persistence guarantees with `psync` is possible, by enabling a `fsync` call after each pwrite. However, completion times were so long that we did not include them in these plots (more than an hour for 20 GiB of pure writes).

We expected NVPages to be less efficient in pure write workloads, because the use of redo logs leads to write every data to NVMM twice. On the other hand, we also expected it to be more efficient than NVLog on mixed IO workloads, because it can store much more data in its page cache, increasing the cache hit probability and reducing interactions with the SSD to the minimum.

However, these results show NVLog performs significantly better in almost every workload. The pure read performance of NVPages reveals a fundamental flaw that prevents it to perform better with mixed IOs. By design, cache misses have a cost in NVPages, because they imply to copy the missing page to NVMM. But the main flaw in this design is the bandwidth limitation of current NVMM compared to DRAM. NVPages can take pages from the LPC in DRAM, but will then require to read in NVMM to retrieve them for reads or writes. On the

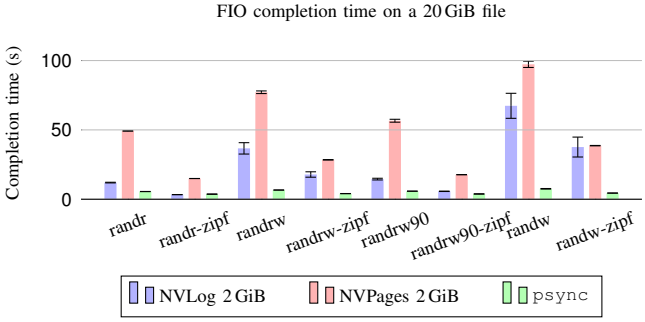

Fig. 3. FIO benchmarks with 2 GiB of NVMM cache

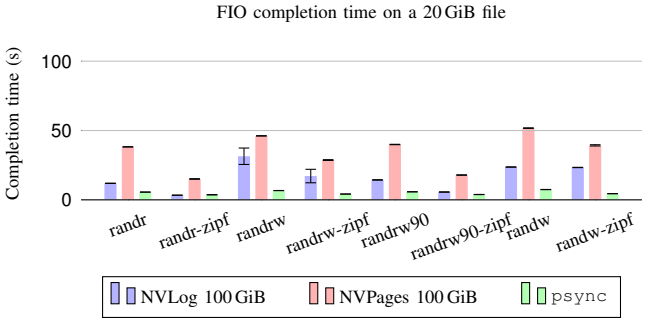

Fig. 4. FIO benchmarks with 100 GiB of NVMM cache

contrary, NVLog keeps fresh pages in DRAM, which allows us to get the full potential of DRAM read bandwidth, as we measured in Fig. 3 and Fig. 4 with `randr` and `randr-zipf` benchmarks.

IV. Conclusion and Future Work

In its current state, NVLog seems to have a clear edge over NVPages. It performed better on all workloads, even on those we expected NVPages to be more efficient. That said, some additional logic should be added to both caches implementations in order to evaluate their performance on multithread workloads. From a design point of view, NVPages has several advantages and may outperform NVLog on parallel IOs thanks to its independent redo logs (while NVLog must share the same log with all threads). Furthermore, the main bottleneck we found in NVPages relies on the difference of performance between DRAM and NVMM.

References

[1] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module," 2019.

[2] R. Dulong, R. Pires, A. Correia, V. Schiavoni, P. Ramalhete, P. Felber, and G. Thomas, "NVCache: A Plug-and-Play NVMM-based I/O Booster for Legacy Systems," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 186–198.

[3] "Intel 3D XPoint™ Technology," https://intel.ly/2XBUk4M, 2019.

[4] J. Axboe, "Fio-flexible I/O tester synthetic benchmark," https://github.com/axboe/fio, accessed: 2023-01-5.

[5] "Zipfian law," https://en.wikipedia.org/wiki/Zipf%27s_law, accessed: 2023-01-5.