NVCache : A plug-and-play NVMM-based IO booster for legacy systems

Rémi Dulong [1] Rafael Pires [2] Andreia Correia [1]
Valerio Schiavoni [1] Pedro Ramalhete [3] Pascal Felber [1]
Gaël Thomas [4]

[1]University of Neuchâtel, Switzerland
[2]Swiss Federal Institute of Technology in Lausanne, Switzerland
[3]Cisco Systems
[4]Telecom SudParis/Institut Polytechnique de Paris

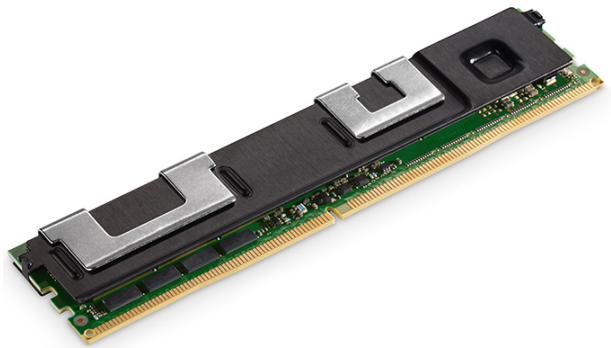
51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks(DSN 2021)

Tuesday, June 22nd

What is NVMM?

NVMM = ***Non-Volatile Main Memory***

- Fast & byte addressable (as RAM)
- Persistent (as an SSD)



512 GB of Intel Optane DCPMM

Intel Optane DCPMM performances

4 kB direct random writes :

	DDR4 DRAM[0]	Intel Optane[1]	SSD	HDD
Avg. Bandwidth	2.2 GB/s	790 MB/s	90 MB/s	1.5 MB/s
Agv. Latency	1.4 µs	5 µs	45 µs	4000 µs
Typical capacity	32 GB	128 - 512 GB	Some TB	4 - 12 TB
Price/GB	12 - 15 $/GB	4.5 - 13 $/GB	0.2 $/GB	0.1 $/GB

[1] With tmpfs

[2] ext4 (DAX)

How to use NVMM?

1. Use PMDK : Persistent Memory Development Kit

How to use NVMM?

1. Use PMDK : Persistent Memory Development Kit
2. Use a DAX (Direct Access) file system

How to use NVMM?

1. Use PMDK : Persistent Memory Development Kit
2. Use a DAX (Direct Access) file system
3. Mmap `/dev/dax1.0` ?

How to use NVMM?

1. Use PMDK : Persistent Memory Development Kit
2. Use a DAX (Direct Access) file system
3. Mmap `/dev/dax1.0` ?
4. Something else?

The idea of NVCache

We want :

- Crash resilience

The idea of NVCache

We want :

- Crash resilience
- Performance

The idea of NVCache

We want :

- Crash resilience
- Performance
- Legacy softwares support

The idea of NVCache

We want :

- Crash resilience
- Performance
- Legacy softwares support
- Legacy file systems support

The idea of NVCache

We want :

- Crash resilience
- Performance
- Legacy softwares support
- Legacy file systems support

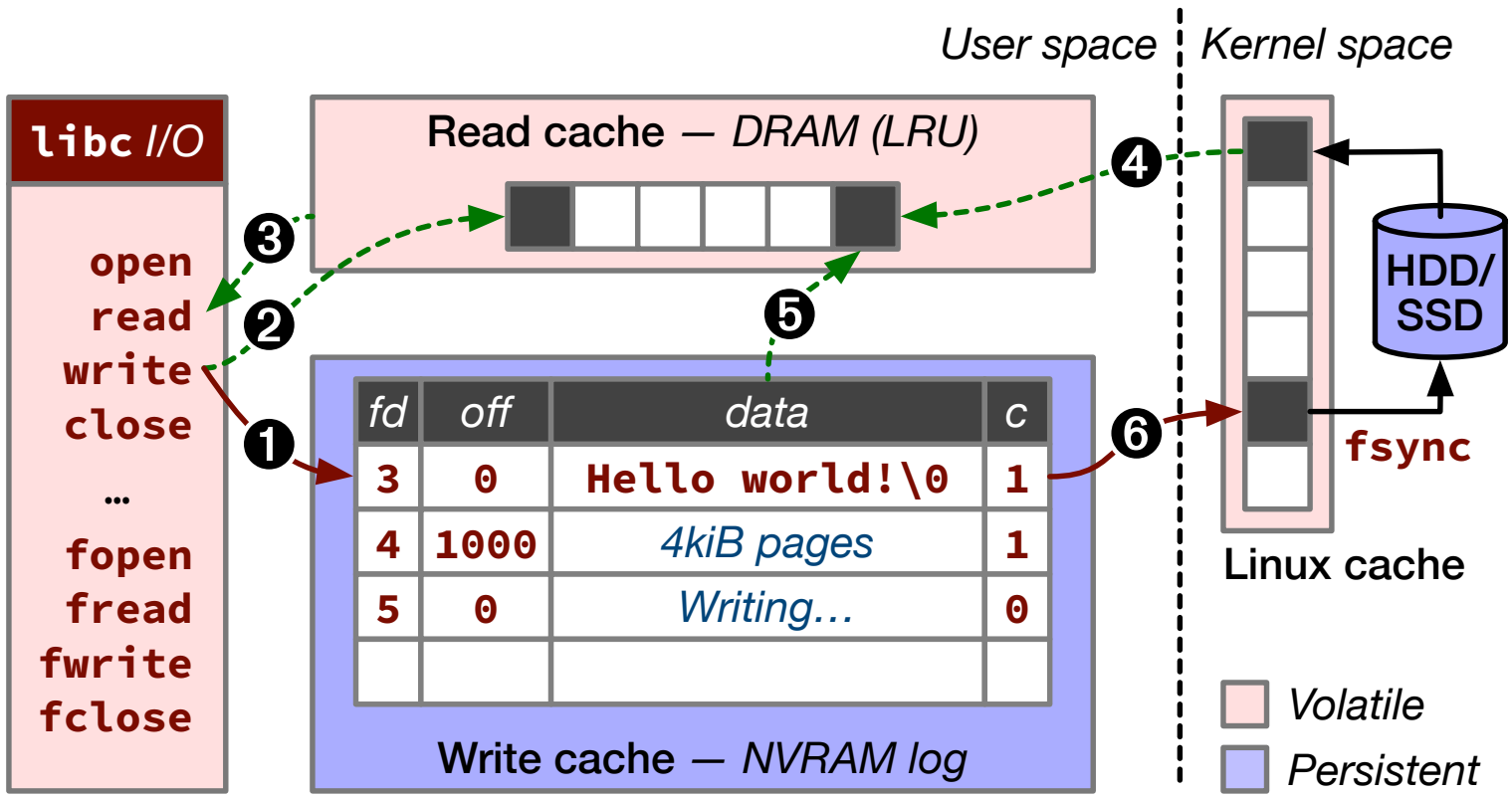
$\Rightarrow$ **Persistent Write cache**

Implementation

Execution

- A fork of the musl libc
- Modifications of *read()*, *write()*, *fsync()*, etc. . .
- Replaces the system libc in Alpine Docker containers

Architecture



Code example

Without NVCache:

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(){

  char *strs[7] = {"a", "b", "c", "d", "e", "f", "g"};
  int fd = open("testfile", O_CREAT|O_RDWR);

  for(int i=0; i<7; i++){
    write(fd, strs[i], 1);
  }
  fsync(fd);
  // =========> Persistence guarantee
  close(fd);
}
```

Code example

Without NVCache:

```c
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(){

  char *strs[7] = {"a", "b", "c", "d", "e", "f", "g"};
  int fd = open("testfile", O_CREAT|O_RDWR);

  for(int i=0; i<7; i++){
    write(fd, strs[i], 1); // Crash?
  }
  fsync(fd);
  // =========> Persistence guarantee
  close(fd);
}
```

Code example

With NVCache:

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(){

  char *strs[7] = {"a", "b", "c", "d", "e", "f", "g"};
  int fd = open("testfile", O_CREAT|O_RDWR);

  for(int i=0; i<7; i++){
    write(fd, strs[i], 1);
  // ========================> Persistence guarantee
  }
  fsync(fd); // Does nothing
  close(fd); // Flushes NVM => Disk
}
```

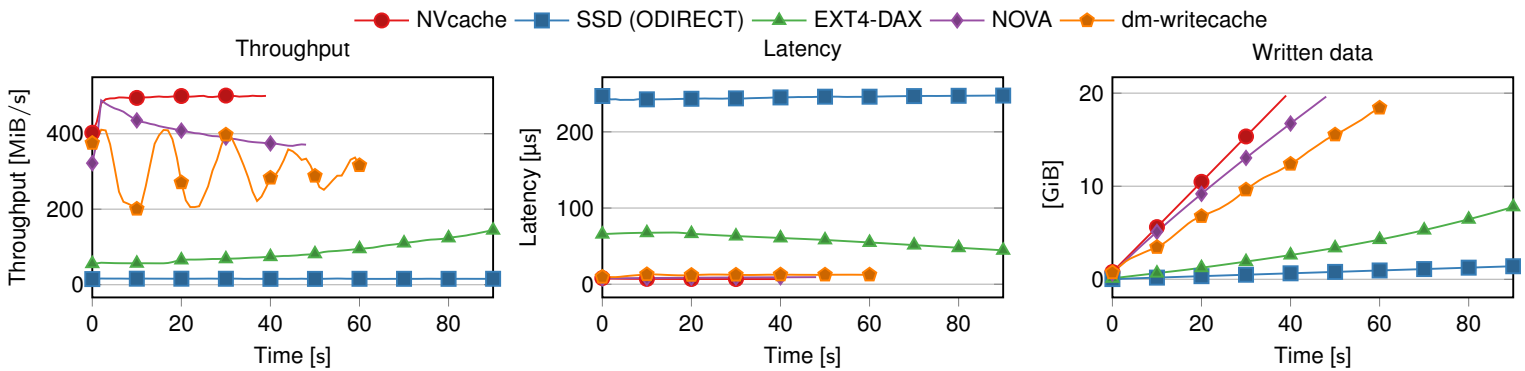
Code example

With NVCache:

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(){

  char *strs[7] = {"a", "b", "c", "d", "e", "f", "g"};
  int fd = open("testfile", O_CREAT|O_RDWR);

  for(int i=0; i<7; i++){
    write(fd, strs[i], 1); // Crash?
  // ========================> Persistence guarantee
  }
  fsync(fd); // Does nothing
  close(fd); // Flushes NVM => Disk
}
```

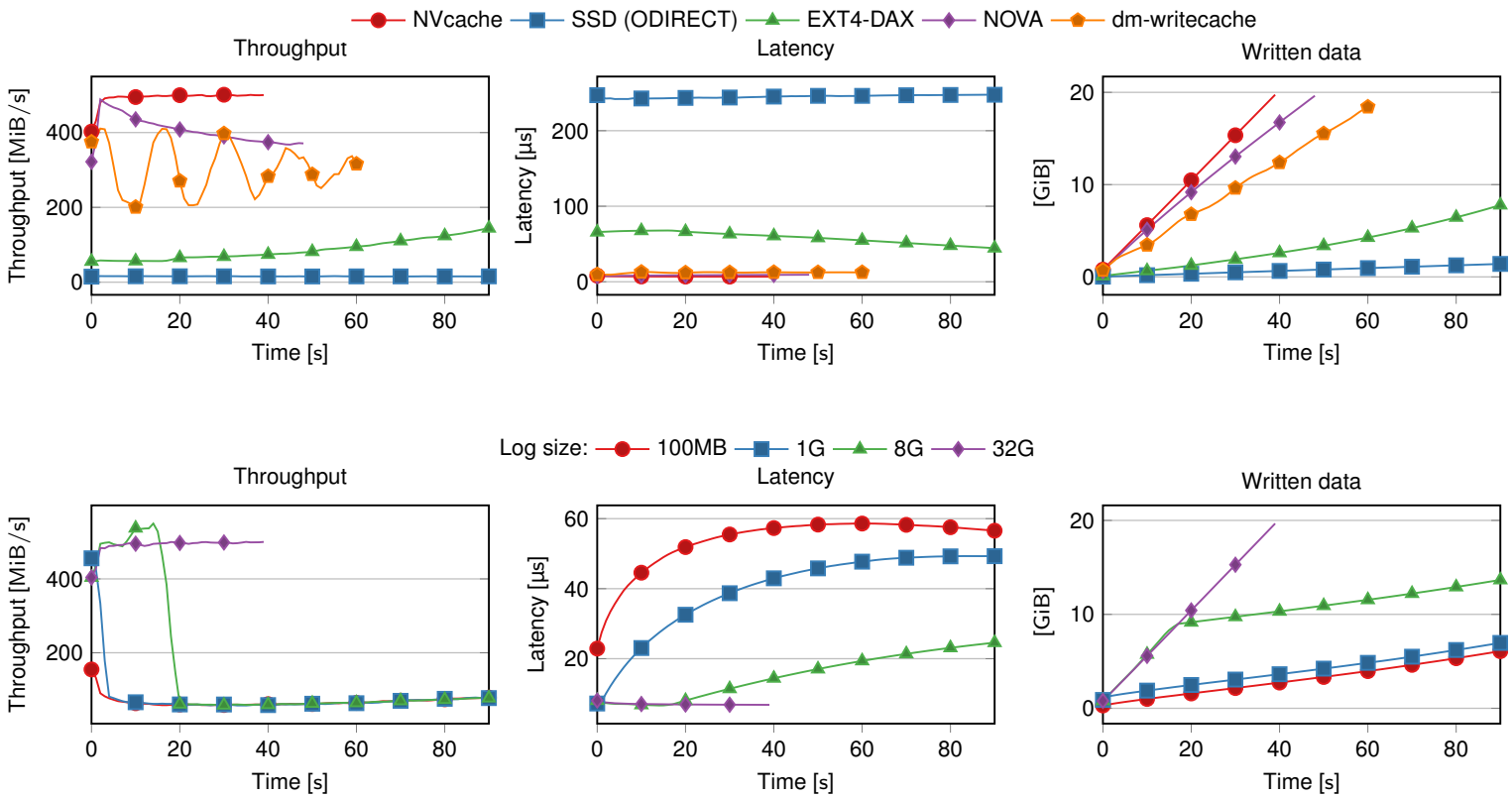
Benchmarks

4 KiB random writes



Benchmarks

4 KiB random writes



NVCache: Conclusion

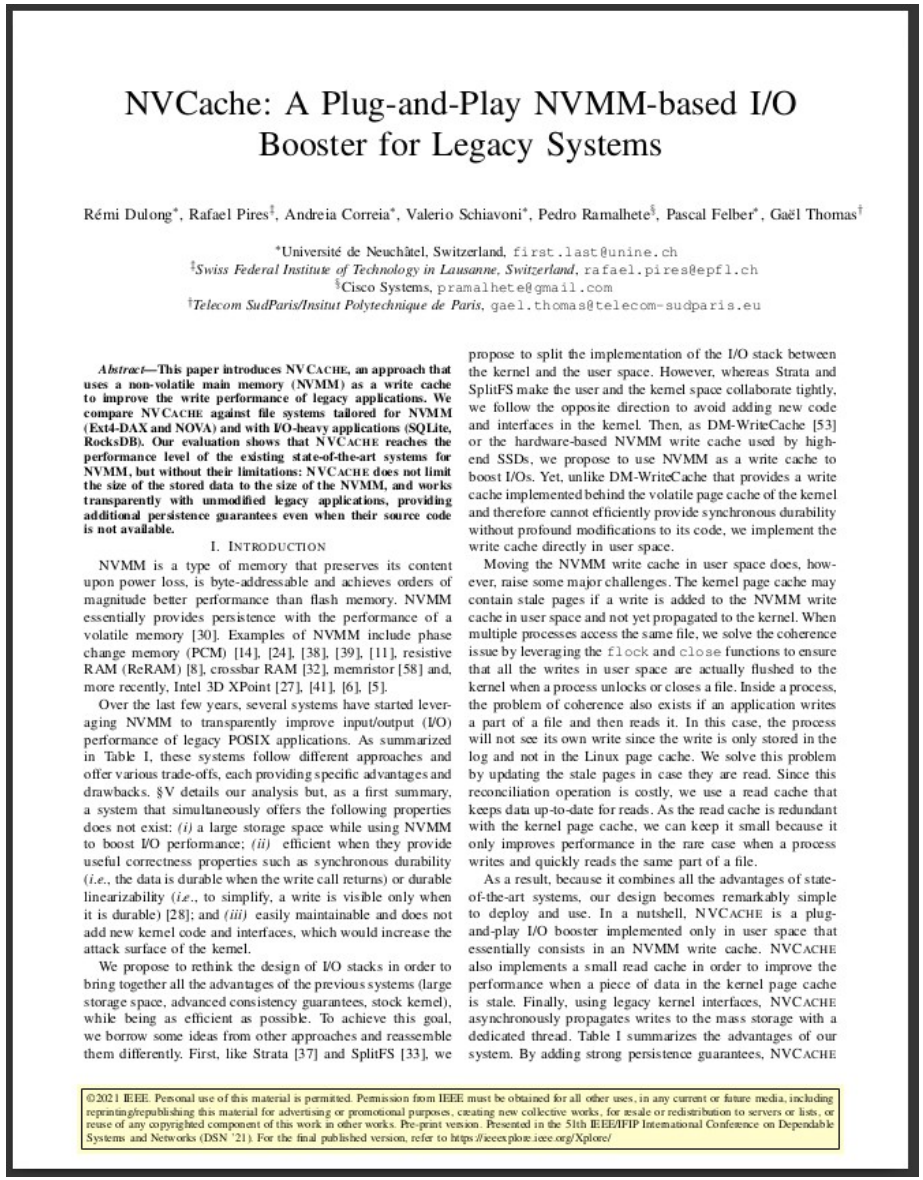
We managed to:

- Add new guarantees
- Keep good performances
- Exceed the limited NVM capacity

$\Rightarrow$ **Less than 3000 lines of code**

Thank you for your attention!

Questions?



NVCache: A Plug-and-Play NVMM-based I/O Booster for Legacy Systems

Rémi Dulong*, Rafael Pires‡, Andreia Correia*, Valerio Schiavoni*, Pedro Ramalhete§, Pascal Felber*, Gaël Thomas†

*Université de Neuchâtel, Switzerland, first.last@unine.ch
‡Swiss Federal Institute of Technology in Lausanne, Switzerland, rafael.pires@epfl.ch
§Cisco Systems, pramalhete@gmail.com
†Telecom SudParis/Insitut Polytechnique de Paris, gael.thomas@telecom-sudparis.eu

***Abstract*—This paper introduces NVCache, an approach that uses a non-volatile main memory (NVMM) as a write cache to improve the write performance of legacy applications. We compare NVCache against file systems tailored for NVMM (Ext4-DAX and NOVA) and with I/O-heavy applications (SQLite, RocksDB). Our evaluation shows that NVCache reaches the performance level of the existing state-of-the-art systems for NVMM, but without their limitations: NVCache does not limit the size of the stored data to the size of the NVMM, and works transparently with unmodified legacy applications, providing additional persistence guarantees even when their source code is not available.**

I. Introduction

NVMM is a type of memory that preserves its content upon power loss, is byte-addressable and achieves orders of magnitude better performance than flash memory. NVMM essentially provides persistence with the performance of a volatile memory [30]. Examples of NVMM include phase change memory (PCM) [14], [24], [38], [39], [11], resistive RAM (ReRAM) [8], crossbar RAM [32], memristor [58] and, more recently, Intel 3D XPoint [27], [41], [6], [5].

Over the last few years, several systems have started leveraging NVMM to transparently improve input/output (I/O) performance of legacy POSIX applications. As summarized in Table I, these systems follow different approaches and offer various trade-offs, each providing specific advantages and drawbacks. §V details our analysis but, as a first summary, a system that simultaneously offers the following properties does not exist: *(i)* a large storage space while using NVMM to boost I/O performance; *(ii)* efficient when they provide useful correctness properties such as synchronous durability (*i.e.*, the data is durable when the write call returns) or durable linearizability (*i.e.*, to simplify, a write is visible only when it is durable) [28]; and *(iii)* easily maintainable and does not add new kernel code and interfaces, which would increase the attack surface of the kernel.

We propose to rethink the design of I/O stacks in order to bring together all the advantages of the previous systems (large storage space, advanced consistency guarantees, stock kernel), while being as efficient as possible. To achieve this goal, we borrow some ideas from other approaches and reassemble them differently. First, like Strata [37] and SplitFS [33], we propose to split the implementation of the I/O stack between the kernel and the user space. However, whereas Strata and SplitFS make the user and the kernel space collaborate tightly, we follow the opposite direction to avoid adding new code and interfaces in the kernel. Then, as DM-WriteCache [53] or the hardware-based NVMM write cache used by high-end SSDs, we propose to use NVMM as a write cache to boost I/Os. Yet, unlike DM-WriteCache that provides a write cache implemented behind the volatile page cache of the kernel and therefore cannot efficiently provide synchronous durability without profound modifications to its code, we implement the write cache directly in user space.

Moving the NVMM write cache in user space does, however, raise some major challenges. The kernel page cache may contain stale pages if a write is added to the NVMM write cache in user space and not yet propagated to the kernel. When multiple processes access the same file, we solve the coherence issue by leveraging the `flock` and `close` functions to ensure that all the writes in user space are actually flushed to the kernel when a process unlocks or closes a file. Inside a process, the problem of coherence also exists if an application writes a part of a file and then reads it. In this case, the process will not see its own write since the write is only stored in the log and not in the Linux page cache. We solve this problem by updating the stale pages in case they are read. Since this reconciliation operation is costly, we use a read cache that keeps data up-to-date for reads. As the read cache is redundant with the kernel page cache, we can keep it small because it only improves performance in the rare case when a process writes and quickly reads the same part of a file.

As a result, because it combines all the advantages of state-of-the-art systems, our design becomes remarkably simple to deploy and use. In a nutshell, NVCache is a plug-and-play I/O booster implemented only in user space that essentially consists in an NVMM write cache. NVCache also implements a small read cache in order to improve the performance when a piece of data in the kernel page cache is stale. Finally, using legacy kernel interfaces, NVCache asynchronously propagates writes to the mass storage with a dedicated thread. Table I summarizes the advantages of our system. By adding strong persistence guarantees, NVCache

Authors

Rémi Dulong
University of Neuchâtel

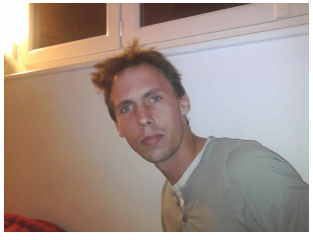
Rafael Pires
University of Neuchâtel

Andreia Correia
University of Neuchâtel

Pedro Ramalhete
Cicso systems

Pascal Felber
University of Neuchâtel

Gaël Thomas
Télécom SudParis



Valerio Schiavoni
University of Neuchâtel

End

Contact : Rémi Dulong, remi.dulong@unine.ch