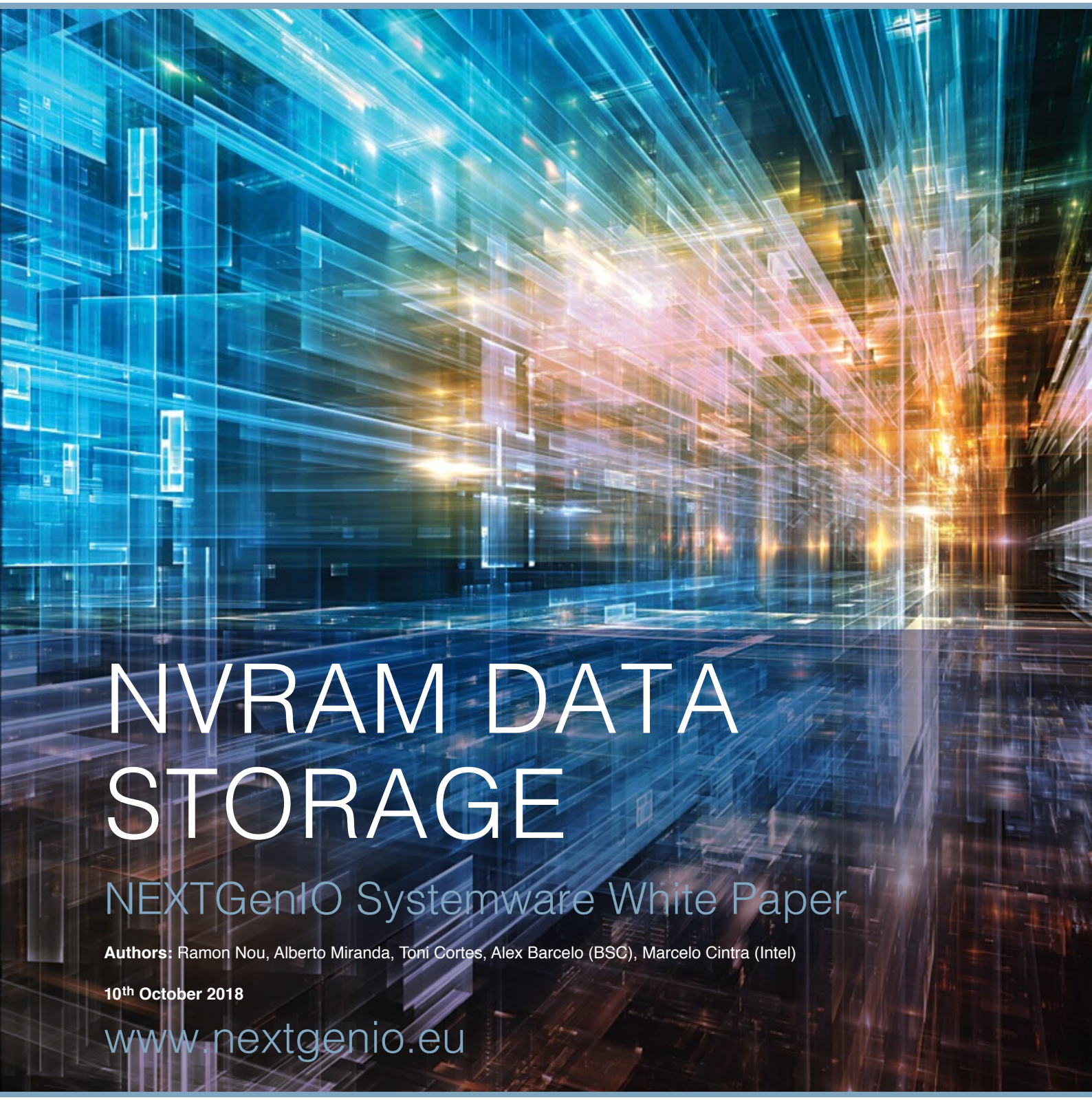


NVRAM DATA STORAGE

NEXTGenIO Systemware White Paper

Authors: Ramon Nou, Alberto Miranda, Toni Cortes, Alex Barcelo (BSC), Marcelo Cintra (Intel)

10th October 2018

www.nextgenio.eu

Next Generation I/O for the Exascale
This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 671951

Contents

Table of Contents

Table of Figures

Foreword

BY DR MICHÈLE WEILAND, NEXTGENIO PROJECT MANAGER

NEXTGenIO is working on improved I/O performance for HPC and data analytics workloads. The project is building a prototype hardware system with byte-addressable persistent memory on the compute nodes, as well as developing the systemware stack that will enable the transparent use of this memory.

Another key outcome from the project's research is the publication of a series of White Papers, covering topics ranging from the architecture requirements and design, to the systemware layer and applications. Each White Paper addresses one of the core challenges or developments that were addressed as part of NEXTGenIO.

This White Paper describes the systemware components that allow applications to store data in the persistent memory layer.

About nextgenio

Current HPC systems perform on the order of tens to hundreds of petaFLOPs. Although this already represents one million billion computations per second, more complex demands on scientific modelling and simulation mean even faster computation is necessary. The next step is Exascale computing, which is up to 1000x faster than current Petascale systems. Researchers in HPC are aiming to build an HPC system capable of Exascale computation by 2022.

One of the major roadblocks to achieving this goal is the I/O bottleneck. Current systems are capable of processing data quickly, but speeds are limited by how fast the system is able to read and write data. This represents a significant loss of time and energy in the system. Being able to widen, and ultimately eliminate, this bottleneck would significantly increase the performance and efficiency of HPC systems.

NEXTGenIO will solve the problem by bridging the gap between memory and storage. This will use Intel's revolutionary new Optane DC Persistent Memory, which will sit between conventional memory and disk storage. NEXTGenIO will design the hardware and software to exploit the new memory technology. The goal is to build a system with 100x faster I/O than current HPC systems, a significant step towards Exascale computation.

The advances that Optane DC Persistent Memory and NEXTGenIO represent are transformational across the computing sector.

1 Executive Summary

This White Paper discusses some of the software layers being developed to support and exploit NVDIMMs, from the driver to the OS and the applications. Those layers, collectively known as systemware, are the ones that can be used from the runtime system, schedulers and applications. In this document, we present the final software prototype of the different software layers, including the API that will be available to users. We have conducted tests on pre-release hardware to detect potential issues early.

The contents cover the following file systems:

1) We present the final implementation of **echofs**, a node-local filesystem design to efficiently exploit NVDIMMS, with some performance numbers with available hardware.
2) The different modes and final release of **dataClay**, an object store, including some performance numbers comparing SSDs (based on NAND) and Intel® Optane™ devices (based on 3D XPoint).
3) A new storage option called **GekkoFS** created in collaboration with Johannes Gutenberg University Mainz (JGU). Partners in NEXTGenIO have implemented POSIX capabilities for this filesystem.

2 Introduction

The addition of NVDIMMs based on the new 3D XPoint™ technology from Intel makes NEXTGenIO different from other architectures. NVDIMMs based on the new 3D XPoint™ technology support different modes and new storage paradigms which require research with respect to different applications. The purpose of this White Paper is to describe the view that the OS and the applications have of the NVRAM layer, while describing the storage paradigms and filesystems that are available in the project.

2.1 Glossary of Acronyms

Acronym	Description
1LM	1Lm refers to an nvdimm mode with only one level of memory. This represents the mode where dram and nvram are used as separate memory structures, with their own memory address spaces. This means that it is possible to address all the dram and nvram from an application, given a total memory space of the total dram plus the total nvram installed in a node. 1Lm allows persistent memory instructions to be issued and supports block persistent mode and load/store instructions as well (volatile mode).
2LM	2Lm refers to an nvdimm mode providing two memory levels. This represents the mode where the dram is used as a transparent memory-side cache for the nvram in the system. Applications do not see the dram memory address space, only the nvram memory address space. This means the total memory available to applications is the size of the total nvram memory in the node (minus any block persistent storage that has been created). 2Lm cannot issue persistent memory instructions, it can only be used in load/store (volatile) mode. However, it can co-exist with some amount of block-persistent storage.
3D XPoint™	This refers to the intel® 3d xpoint™ memory technology: a simple, stackable, and transistor-less memory design that creates fast, inexpensive, and non-volatile storage memory devices with low latency [1].
API	Application programming interface.
COMPSs	Comp superscalar: parallel programming model for distributed infrastructures
DAX	Direct access extensions, enables storing and reading data direct to devices bypassing kernel mechanism (page-cache, buffering)
DRAM	A random-access memory (ram) that uses a single transistor-capacitor pair to store each bit. “D” stands for “dynamic, since dram cells have to be refreshed at regular intervals to avoid losing their charge (and thus the stored information).
Extrae	Package devoted to generating paraver trace-files
FUSE	Filesystem in userspace. Extensions to the kernel to create filesystems without kernel modifications
JSON	Javascript object notation, an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types (or any other serialisable value).
LLC	Last level cache, the bottom level of a cache hierarchy; entries are loaded and written back from/to dram.
MAP	Performance monitoring tool for parallel, multithreaded or single threaded c++, fortran codes, developed by partners arm/allinea.
Memory-Side Cache	A memory-side cache holds blocks recently requested from memory. It may be viewed as another level of the memory hierarchy.
Mmap	Memory mapped. Method to map a file contents in memory to accommodate for memory mapped file i/o.

MPI	Message passing interface. A standard for message-based distributed parallel computing.
NVDIMM	Dual inline memory module devices containing nvram.
NVRAM	Non-volatile memory is capable of retrieving stored data even after a power outage. Examples of non-volatile memory include 3d nand flash, solid-state drives, and 3d xpoint™ technology.
OID	Object id, an identifier uniquely referring to an object
OTF2	Open trace format 2. Interface to read and write trace data used by score-p.
Paraver	Open source and trace-based performance analysis tool.
PAPI	Performance application programming interface. A portable interface to query performance data from cpu hardware counters.
PFS	Parallel filesystem
Profile	A performance profile is a table or chart displaying the distribution of one or more performance metrics over the observed program's parts or processes.
PyCOMPs	Python binding of compss.
POSIX	Portable operating system interface. Standard specified by ieee for maintaining compatibility between the operating systems.
Score-P	Performance instrumentation and measurement infrastructure.
Timeline	A timeline-based visualisation reconstructs the initially recorded program states along an increasing time axis, thus, exactly showing when the program did what.
Trace	A program trace is a log of recorded program states together with a time stamp.
Vampir	Open source and trace-based performance monitoring tool for parallel, multithreaded or single threaded c, c++, fortran codes.
YAML	“Yaml ain't markup language” – a human-readable data serialisation language, most commonly used for configuration files.
DCPMM	Data center persistent main memory intel acronym for the nvdimm-p product
B-APM	Byte-addressable random access persistent memory (b-apm)

3 Architecture

The systemware created provides different ways for applications and users to manage the NVDIMM modules (Figure 1). The nodes will be managed using Node Management tools and SLURM, with a set of scripts formatting and preparing the different modes specified by the job description. In this section we will explain the different modes along with the different APIs that will be available.

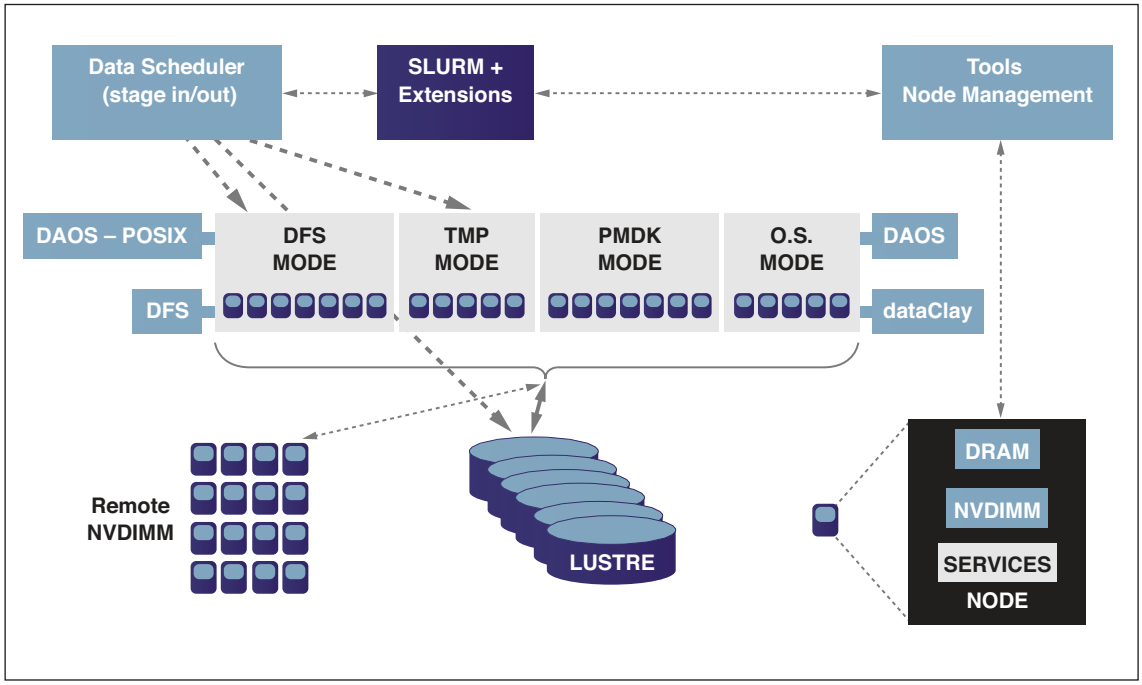


Figure 1: System software architecture diagram.

3.1 NVRAM Libraries

System or application software can use several different NVDIMM libraries, such as libmemkind or PMDK. For the work described in the white paper we are using PMDK libraries to provide the needed persistency for the write operations to the NVRAM space.

3.1.1 libmemkind

The libmemkind library[1] is an open source (BSD license), user-extensible heap manager built on top of jemalloc[2] that enables partitioning of the heap between kinds of memory. The original intended “kind” of memory corresponded to memory systems with NUMA (Non-Uniform Memory Access) behaviour, i.e. where memory latency depends on how close the memory is to the CPU. Additional types of memory are now supported, including a PMEM type that corresponds to memory allocated in non-volatile, byte-addressable media, such as NVDIMMs.

[1] http://memkind.github.io/memkind/
[2] http://jemalloc.net/

The PMEM memory kind provides a higher-level interface for application developers to manage a heap allocated in NVRAM in the persistent memory. When using this memory kind, the library relies on a persistent memory aware file system. It allows programmers to easily create and destroy NVRAM memory allocations and allow for familiar C/C++ style memory allocation (e.g. malloc and free) within such allocations. Figure 2 shows some of the heap management functions provided by libmemkind as of the writing of this document. These are analogous to their *glibc* counterparts with the only addition of a memkind_t identifier of the kind of memory.

```
void *memkind_malloc(memkind_t kind, size_t size);
void *memkind_calloc(memkind_t kind, size_t num, size_t size);
void *memkind_realloc(memkind_t kind, void *ptr, size_t size);
int memkind_posix_memalign(memkind_t kind, void **memptr, size_t alignment, size_t size);
void memkind_free(memkind_t kind, void *ptr);
```

Figure 2: Abridged heap management API of the libmemkind library.

The particular API for the PMEM memory kind is shown in Figure 3 The API mainly provides mechanisms for programmers to create and destroy persistent memory areas on NVRAM.

```
int memkind_pmem_create(struct memkind *kind, const struct memkind_ops *ops, const char
        *name);
int memkind_pmem_destroy(struct memkind *kind);
void memkind_pmem_mmap(struct memkind *kind, void *addr, size_t size);
int memkind_pmem_get_mmap_flags(struct memkind *kind, int *flags);
int memkind_pmem_get_size(struct memkind *kind, size_t *total, size_t *free);
```

Figure 3:

3.1.2 PMDK

While the libmemkind library (Section 3.1.1) provides a convenient and familiar interface for programmers to manage data in a heap allocated in NVRAM, it is not meant as a mechanism for managing data in persistent partitions. The Open Source PMDK collection of targeted libraries[3] offers some of this needed functionality. These libraries are meant to give programmers the ability to instantiate data structures that are both persistent and directly byte-accessible via loads and stores from user programs. This is a new paradigm – Persistent In-Memory Data-Structures – that avoids both explicit serialisation/de-serialisation of data to/from block storage as well as OS paging of memory-mapped files. Like libmemkind, it relies on a persistent memory aware file system and a persistent NVRAM partition.

More specifically, PMDK contains libpmem, providing low-level persistent memory support such as pmem_map to allocate a persistent memory region and pmem_sync for making sure data written to the persistent memory reaches the NVDIMMs[4]. To facilitate the use of persistent in-memory data-structures, PMDK also contains libpmemobj, which allows programmers to create objects in persistent memory and provides mechanisms for transactional updates. The latter is critical to avoid inconsistency in data structures in the presence of system failures during data updates. Transactional updates are implemented using persistent logs. Alternatively, programmers can also create their own transactions and logs using the libpmemlog library.

[3] http://pmem.io/

[4] This is needed since completion of stores from the CPU standpoint does not guarantee persistence of the data in the NVDIMMs since data may still be percolating through the various caches and buffers in the system.

Figure 4 shows some of the key functions provided by libpmem to manage persistent regions and the persistence of changes. Figure 5 shows some of the key functions provided by libpmemobj to manage object storage, to create objects, to persist object changes, to manage persistent locks, to manage lists of objects, as well as to manage transactions. Finally, Figure 6 shows some of the key functions provided by libpmemlog to manage a persistent log.

```
int pmem_unmap(void *addr, size_t len);
int pmem_is_pmem(const void *addr, size_t len); void pmem_persist(const void *addr, size_t
len); int pmem_msync(const void *addr, size_t len);
void *pmem_map_file(const char *path, size_t len, int flags, mode_t mode, size_t *mapped_
lenp, int *is_pmemp);
void pmem_flush(const void *addr, size_t len); void pmem_drain(void);
```

Figure 4: Abridged API of the libpmem library.

```
PMEMobjpool *pmemobj_open(const char *path, const char *layout);
PMEMobjpool *pmemobj_create(const char *path, const char *layout, size_t poolsize, mode_t
mode);
void pmemobj_close(PMEMobjpool *pop);
int pmemobj_alloc(PMEMobjpool *pop, PMEMoid *oidp, size_t size, uint64_t type_num,
pmemobj_constr constructor, void *arg);
void pmemobj_free(PMEMoid *oidp);
void pmemobj_persist(PMEMobjpool *pop, const void *addr, size_t len);
void pmemobj_flush(PMEMobjpool *pop, const void *addr, size_t len);
void pmemobj_drain(PMEMobjpool *pop);
int pmemobj_mutex_lock(PMEMobjpool *pop, PMEMmutex *mutexp);
int pmemobj_mutex_trylock(PMEMobjpool *pop, PMEMmutex *mutexp);
int pmemobj_mutex_unlock(PMEMobjpool *pop, PMEMmutex *mutexp);
int pmemobj_list_insert(PMEMobjpool *pop, size_t pe_offset, void *head, PMEMoid dest, int
before, PMEMoid oid);
int pmemobj_list_remove(PMEMobjpool *pop, size_t pe_offset, void *head, PMEMoid oid, int
free);
int pmemobj_tx_begin(PMEMobjpool *pop, jmp_buf *env, enum pobj_tx_param, ...);
void pmemobj_tx_abort(int errnum);
void pmemobj_tx_commit(void);
int pmemobj_tx_end(void);
```

Figure 5: Abridged API of libpmemobj library.

```
PMEMlogpool *pmemlog_open(const char *path);
PMEMlogpool *pmemlog_create(const char *path, size_t poolsize, mode_t mode);
void pmemlog_close(PMEMlogpool *plp);int pmemlog_append(PMEMlogpool *plp, const void
*buf, size_t count);
void pmemlog_rewind(PMEMlogpool *plp);
void pmemlog_walk(PMEMlogpool *plp, size_t chunksize, int (*process_chunk)(const void *buf,
size_t len, void *arg), void *arg);
```

Figure 6: Abridged API of the libpmemlog library.

3.2 Developed and Adapted Filesystems

Firstly, we created echofs, a POSIX-like filesystem which uses the NVRAM of the compute nodes of the NEXTGenIO prototype in a transparent way. The main objective was to be able to use legacy applications and to preload files residing on an external Parallel FileSystem (PFS) automatically into the NVRAM so that applications can make best use of the technology.

The second development is a modification of the dataClay object store. dataClay is an object store created considering NVRAM as it allows to store both data objects and methods operating on these objects together. dataClay can take advantage of any byte-addressable storage. Developed initially for traditional storage systems, we created three versions that allow users to take advantage of the compute nodes NVRAM in different ways. We will explain the three versions and present their strengths and weaknesses of each one in section 3.4.1.

Finally, and to complete the POSIX compatibility, we include GekkoFS, a joint development with Johannes Gutenberg University in Mainz, that offers an ephemeral file system.

3.3 Block Level Storage

3.3.1 Standard FS – echofs

The current growth in data-intensive scientific applications poses strong demands on the HPC storage subsystem, since data needs to be typically copied from compute nodes to I/O nodes and vice versa when calculations start and stop. In this scenario, the emerging trend of adding denser, NVRAM-based storage to compute nodes as burst buffers, offers the possibility of using these resources to construct temporary filesystems that perform ad-hoc I/O optimisations for specific batch jobs. Echofs coordinates with the job scheduler to preload a job's input files into each compute node's NVRAM.

The developed version of the echofs filesystem includes single node support and it allows to transparently move files from the PFS to the NVRAM (stage-in) and from NVRAM to PFS (stage-out). The only modifications needed are in the SLURM job description file, which requires to define the directories that the user is going to use.

echofs is available at **https://github.com/NGIOproject/echofs**

3.3.1.1 Design

The Echo Filesystem (echofs (1)) is designed as a temporary filesystem layer that resides on compute nodes between applications and the centralised parallel filesystem (PFS), and that survives for as long as a batch job (or a series of related batch jobs) needs it. This additional storage layer is added to the compute nodes' I/O stack with two main goals in mind:

(1) To hide the complexity of the diverse levels of storage that may be available to an application running in a compute node by aggregating them into a virtual storage device with a single mount point.

(2) To automatically manage the migration of data within this hierarchy according to internal algorithms and user-provided hints to balance I/O performance and storage capacity.

[5] https://github.com/daos-stack/daos

When a batch job is submitted, the job scheduler determines the initial set of data dependencies, allocates a set of compute nodes on the cluster, constructs an instance of echofs across these nodes and communicates the captured information to the filesystem. At this point, the filesystem's components running on the compute node prefetch all input data from the PFS. If we are on the distributed version (discussed on the next subsection) the data will be distributed across the compute nodes' NVRAM.

Once the preload process completes, the batch job can start and the application can use the standard POSIX I/O functions to access the data contained in the filesystem. The main reason for supporting the POSIX I/O API is to allow legacy applications to easily use (and benefit from) the available NVRAM buffers with no modifications to their code, since the filesystem interface remains unchanged.

Also, echofs mirrors the namespace of the underlying PFS to allow applications to access any of its preloaded files when the filesystem was mounted. We decided not to allow a transparent traverse to the underlying PFS as it adds complexity to the metadata management. Due to the lifetime of the ad hoc filesystem, we decided that any request can be issued directly to the PFS mount point.

Moreover, any new files created are written directly in the NVRAM buffers and are only flushed to the PFS when the job completes. Also note that this is done only if the user or the job scheduler explicitly request for these files to be persistent, which would allow upcoming jobs to reuse the available data, if needed (e.g. to support a workflow of related jobs that use the same input data, or that rely on the results of a previous batch job). This helps reduce the I/O contention on the PFS, by eliminating the need to persist any temporary files and/or checkpoints generated by an application, at the expense of shifting the responsibility of "tagging" output files to the scheduler/user.

3.3.1.2 Metadata Management

The management of POSIX metadata operations (such as open(), stat(), and readdir(), among others) in a PFS is challenging and often relies on one or several metadata servers that are in charge of keeping metadata information consistent, as well as providing file handles, and giving and revoking rights to the clients to update and cache file information. Therefore, given that metadata operations are in the critical path of many I/O operations, the metadata servers can become a serious bottleneck when receiving concurrent requests from considerable number of clients. Nevertheless, due to its intended usage, the echofs filesystem has several advantages with regards to general purpose PFSs:

- Since its lifetime is limited, the temporary filesystem needs to support only those application processes running in the context of the corresponding batch job, which, when compared to the larger number of clients that a general purpose PFS needs to serve, reduces the scalability requirements considerably.
- Similarly, since echofs is informed by the job scheduler about the files that will be used by the batch job before it starts running, it can control the volume at which these operations should arrive to the PFS. For instance, if it needs to open many files to preload data into the NVRAM buffer (as it often happens with checkpoints writing from and to different nodes), it can throttle these requests to avoid overloading the PFS' metadata subsystem.

3.3.1.3 Data Management

In the prototype, NVRAM accesses are managed using PMDK Library (2) to create memory-mapped files on top of a filesystem with DAX capabilities, which allows for direct access to the hardware with byte granularity. Using this library has the advantage that the same interface can be used for NVM, SSDs and HDDs, as long as the underlying filesystem supports mmap(). Since the PMDK library is intended to work with memory-mapped files, echofs relies on NVRAM segments that represent a contiguous range of file data that is backed by a PMDK-mapped file (see Figure 7). Thus, when a file is preloaded into echofs, a new segment is created and populated with the contents of its PFS counterpart. The segment is then registered into a per-file segment tree which allows to resolve offsets in O(log n) time. When files need to grow (or are created anew), new segments of a pre-configured size are allocated and appended to the tree. This allows files to grow as needed and allows to easily support sparse files to reduce storage overhead. When the filesystem is unmounted, the segments for persistent files are moved and consolidated to the PFS. Segments for reusable files are consolidated into a unified file so that they can be used by future processes. Persistence of write operations is internally managed by echofs by using the appropriate msync() and drain() calls through PMDK.

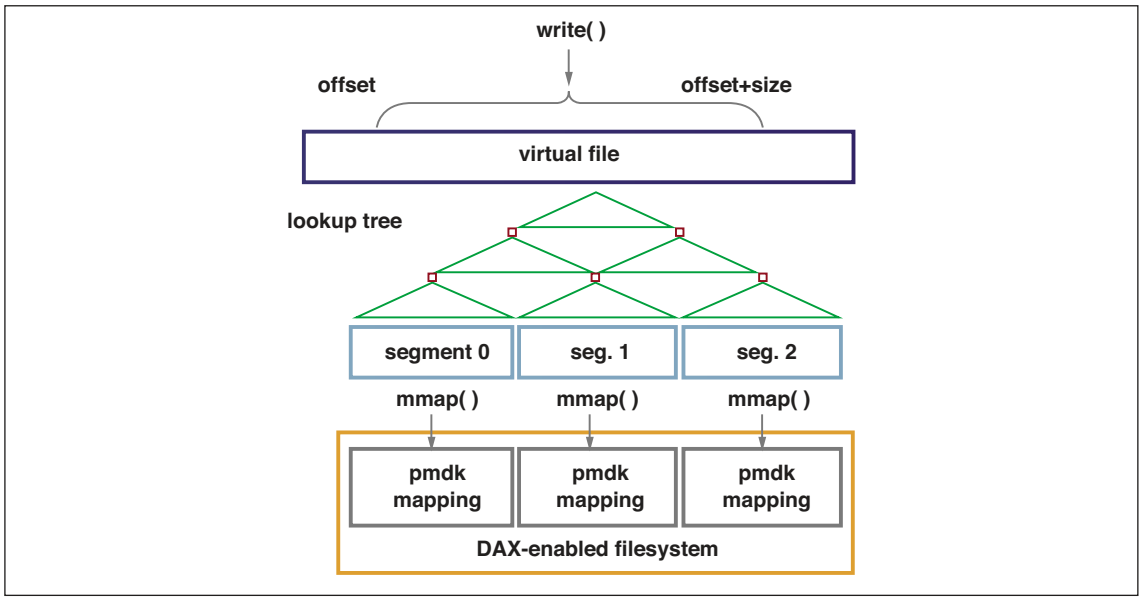


Figure 7: Internal echofs file segments and NVM mappings

3.3.1.4 API and usage

As an example, we can see the next (Figure 8) echofs job configuration file (use with –c)

```
## global settings that affect the behavior of echofs
global-settings: [
  root-dir: "/home/rnou/efs-ng/build/root/",
  mount-dir: "/home/rnou/efs-ng/build/mnt/",
  results-dir: "/home/rnou/efs-ng/build/root/job42/",
]
## definition of backends
backends: [
  [ id: "nvml://",
    type: "NVRAM-NVML",
    capacity: "2 GiB",
    daxfs: "/mnt/pmem/"
   ]
]
## resources to import/export
resources: [
 # resource must be transferred to results-subdir at unmount time
  [ path: "/home/rnou/efs-ng/build/root/file2.tmp",
    backend: "nvml://",
    flags: "persistent"
  ],
#Imported paths recursively
 # resource must be transferred to results-subdir at unmount time
  [ path: "/home/rnou/efs-ng/build/root/recursivepath",
    backend: "nvml://",
    flags: "persistent"
  ],
  # resource can be safely deleted from the backend at unmount
  [ path: "/home/rnou/efs-ng/build/root/fileA.tmp",
    backend: "nvml://",
    flags: "temporary"
  ]
 ]
```

Figure 8: echofs configuration file

src/efs-ng -f -c ../examples/preload.conf

This will mount the echofs on the specified directory of the .conf file, and also preload some files and directories into it. An application can use the mount point as any other filesystem.

The second mode is to use the API, which requires linking the application code with efs-api, however only load and unload are fully implemented at this time.

There is an extra backend able to use a character device directly:

```
[ id: "nvml://",
  type: "NVRAM-DEVDAX",
  capacity: "2 GiB",
  daxfs: "/dev/dax13.0"
]
```

3.3.1.5 Benchmark

In this section, we evaluate echofs to prove how by using it applications can benefit from NVRAM performance without having to deal with the complexity required to manage and access it. We describe the experimental setup and discuss the workloads that we simulate with well-known I/O access microbenchmarks. Then, we evaluate the data and metadata performance of the echofs prototype, as well as a simple coordination mechanism for job schedulers (i.e. SLURM) to optimise stage-in/stage-out procedures in coordination with echofs. Finally, we discuss the measured results.

3.3.1.5.1 Experimental setup

All tests were done on a cluster which consists of 10 nodes interconnected by a 1GbE link. Each node is composed of a 4-core Intel Xeon E5-2609v2 processor (2.50GHz), with 32GiB DRAM, and a 400GB SSD (Intel SSD P3500). A Lustre 2.8 filesystem was configured in the cluster using one node for the Metadata Service (MDS), and two nodes as Object Storage Targets (OSTs), using the aforementioned SSDs as the storage backend. We also include results for a 375 GB Intel® Optane™ SSD unit to get a lower bound of what can be achieved with NVRAM, used with normal filesystems (XFS).

The NVRAM hardware itself is emulated using a DRAM backend, on top of which an ext4 filesystem with DAX support is mounted. Note that even if the backend hardware is actually not NVRAM, this serves to get a rough idea of the kind of maximum throughput to be expected with state-of-the-art APIs, given that the latencies for the more performant NVDIMM devices are expected to be higher than with DRAM (3).

For benchmarking, we selected the IOR[6] and MDTEST microbenchmarks, since they are well-established in the field, allow exercising both metadata and data workloads and are highly configurable.

3.3.1.5.2 Metadata performance

To evaluate the performance of echofs's metadata subsystem we generate a workload for it using the MDTEST microbenchmark and compare its performance against the metadata subsystems of the ext4 (both with a local HDD and a local SSD) and Lustre file systems. Note that even though the echofs prototype performs all metadata operations in RAM, the other filesystems also provide metadata caching to some extent. We run the microbenchmark using the command line mdtest -I 2000000 -b 1 -z 0, which instructs MDTEST to create 2,000,000 files/directory in a single parent directory. We run it with 1 and 4 processes. Since echofs stores metadata in memory, we do not need to evaluate its different configurations, as we will do with IOR. The results (mean), with 10 repetitions, can be found in Table 1.

We observe that, as expected, echofs shows overall better performance than its counterparts, since it has been designed precisely to avoid the metadata issues of shared filesystems such as Lustre, and the latency of RAM-backed metadata should be better than HDD/SSD-backed metadata. The only exception can be observed with stat() operations. For these operations, even though echofs uses RAM, we are limited by a maximum number of operations by second supported by FUSE. Though stat() scales nearly linearly on HDDs, SSDs and Lustre, echofs needs to issue 2x the number of context switches compared to a traditional filesystem, due to the FUSE architecture. Before discovering this issue, we experimented with different optimisations to increase stat() performance, such as using a hash map optimised for queries in detriment of other operations. No major benefits were seen.

[6] https://github.com/daos-stack/daos

	CREATE DIR	STAT DIR	REMOVE DIR	CREATE FILE	STAT FILE	REMOVE FILE
HDD(1P)	1,324	515,096	1,913	1,810	370,508	1,810
HDD (4P)	1,460	1,936,912	1,716	2,067	1,412,165	1,751
SSD (1P)	10,744	524,500	23,534	19,572	368,323	25,944
SSD (4P)	11,409	1,821,136	22,514	21,508	1,382,580	29,174
LUSTRE (1P)	7,184	463,454	15,712	9,396	457,911	12,990
LUSTRE (4P)	9,541	1,824,400	11,643	9,894	1,819,549	10,651
ECHOFS (1P)	30,675	37,746	34,445	20,258	35,778	30,658
ECHOFS (4P)	27,731	84,315	32,294	22,047	81,646	30,257

Table 1: MDTEST results in operations per second.

3.3.1.5.3 Data performance

To evaluate the I/O performance of the echofs prototype, we run several workloads using the IOR microbenchmark. In these experiments, we extend the evaluation to other backend filesystems for echofs and different FUSE versions, given that we observed low performance from ext4+DAX, despite it being the filesystem normally used to test NVM libraries. For this reason, we compare the IOR workloads with the following filesystems that offer DAX support: ext4, ext3, XFS, and ramfs. We also include a baseline with tmpfs for comparison even if it does not support DAX since all its operations are performed in RAM. Moreover, we evaluate echofs with the two different FUSE versions: 2.9 and 3.0, patching the kernel to increase the internal buffer size to 12 MiB.

We run the microbenchmark using the command lines ior -b 1g -a POSIX -t 128k and ior -b 1g -a POSIX -F -t 128k, which instruct IOR to create a file/set of files of 1GiB inside a directory, respectively performing shared and non-shared operations upon them using a 128KiB transfer size and 4 processes. Table 2 summarises the results for different storage devices (bypassing the memory cache) compared against the best-performant echofs results. As expected, echofs obtains more performance than using traditional storage, as it uses a memory-like technology, but also more performance than local SSDs. In the case of Optane, we observe how it surpasses SSD speed but falls short with regard to performance using memory. However, upcoming NVRAM technology (as planned for the NEXTGenIO prototype) is expected to show better performance. As mentioned, we found discrepancies between the echofs results when using different DAX-enabled filesystems as data backends.

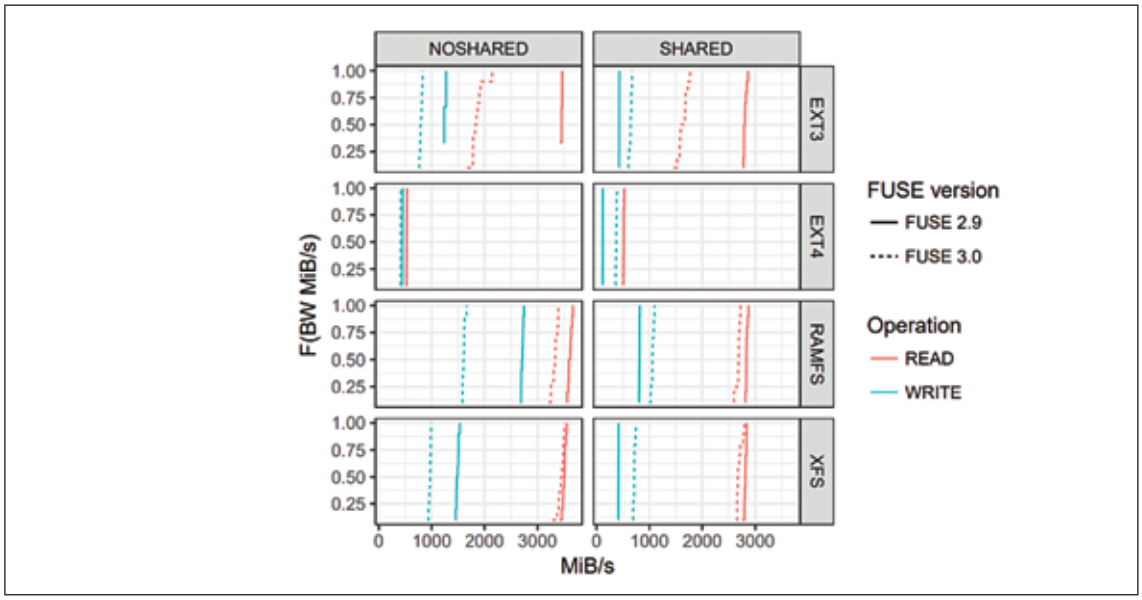


Figure 9: Throughput using different FUSE versions and DAX filesystems.

In Figure 9, we show an ECDF (Empirical Cumulative Distribution Function) of the bandwidth obtained with different DAX filesystems (ramfs, ext3, ext4 and XFS) and FUSE versions used through echofs. XFS and ramfs showed good throughput, but we observed low performance with ext4, one of the de facto filesystems used to test NVM libraries. Also note that the ECDF lines are mostly vertical, which means that bandwidth is fairly stable in all repetitions. As a side note, there is more performance with FUSE 3.0 in shared experiments since the internal write buffer increases, and the number of locks operations is thus reduced.

	SHARED		NON-SHARED	
	Read (MiB/s)	Write (MiB/s)	Read (MiB/s)	Write (MiB/s)
LUSTRE (1GBE)	111.39	112.81	145.39	113.18
HDD	114.93	140.57	89.77	123.78
SSD	235.12	288.97	232.39	291.96
OPTANE	1,035.39	616.43	752.72	733.74
FUSE 2.9 – RAMFS - ECHOFS	2,835.24	807.37	3,606.85	2,710.27

Table 2: IOR results (HDD, SSD, Optane and best performant echofs option)

3.3.1.5.4 Coordinated echofs Stage-in/out vs. Direct IOR in PFS

To prove that using echofs can reduce the overall PFS I/O contention despite the fact that it needs to stage-in and stage-out data from it, we ran a proof-of-concept experiment where three 16GiB files were created in Lustre, which were then sequentially read and rewritten, concurrently from three different nodes (see Table 3). Relying on the PFS, so that data always stays in-sync as it is typically done in HPC clusters, required 835.5 seconds to finish the concurrent operations. Relying on echofs' capabilities to stage-in and stage-out automatically, the same workload took around 715.3 seconds with concurrent stage-in phases (a 14.38% improvement), and 583.2 seconds with a simple SLURM heuristic to interleave stage-in phases (a 30.19% improvement). Benefits were more significant if data was reused. For example, using two nodes and two 2.5GiB files only, required around 240.5 seconds to execute 10 iterations directly on the PFS, but only 71.8 seconds to run them in echofs with data reuse, a 70.14% improvement.

	TIME TO COMPLETION	IMPROVEMENT
PFS I/O 3 x 16GiB	835.5s	---
Uncoordinated 3 x 16GiB	715.3s	14.38%
Interleaved 3 x 16GiB	583.2s	30.19%
PFS I/O 2 x 2.5GiB x 10 its	240.5s	---
echofs+reuse I/O 2 x 2.5GiB x 10 its	71.8s	70.14%

Table 3: Stage-in vs direct PFS I/O results

3.3.1.6 Log Generation

Figure 10 presents the trace format.

timestamp:operation:pid:tid:path\|backend:argument

- **timestamp: epoch with *µ*sec precision**
- **Operation: high/low level operation performed**
  - **o High level:**
    - **Access, close, closedir, create, fsync, open, opendir, read, readdir, stat, truncate, unlink and write**
  - **o Low level**
    - **nvml_read, nvlm_write**
- **pid: PID of the process originating the call**
- **tid: tid of the internal thread serving the call**
- **Arguments: a list of additional fields separated by:**
  - **o High level operations: pathname**
  - **o Read/write operations: pathname: offset, size**
  - **o Nvml_read / nvml_write operations: nvml_address, size**
    - **Address will be 0x0 when reading from a file gap**

Figure 10: echofs log format.

3.3.1.7 Implementation and libraries used

The implementation is done using the FUSE (2.9 and 3.0 supported) file-system in user space kernel module. Although the performance over FUSE may be lower than using kernel interfaces, we decided to use it as it allows us to reduce the efforts need to create very simple functionality and focus on the implementation and research of new functionalities. On the other hand, FUSE needs to be updated to continue being useful in the future given that actual latencies and bandwidth are better than a few years ago.

3.3.1.8 POSIX capabilities implemented

- Permissions are not implemented, as the mount point is only visible by the user.
- Writes and Reads follow POSIX semantics
- Truncate and allocate does not follow POSIX semantics, mainly because truncate will conflict with an ongoing write and reduce their size.

Those features can be revised depending on the applications that we run in NEXTGenIO and if they are an important feature or not.

3.3.1.9 Possible extensions

echofs can leverage the use of filters in the I/O stack to allow deduplication and compression capabilities on the I/O path. For example, we can increase the space available in the NVRAM at the expenses of using more computation, delaying the read/write operations. It is easily done creating a new backend and modifying the put_data/get_data operations on the file.cpp code. As an ad hoc filesystem, such modifications can be done by the application designer.

3.3.1.10 Issues with FUSE

After a deep evaluation we found that FUSE is too slow and cannot compete with a standard filesystem mounted on top of a NVMe device. Metadata and preload options can make a difference, but we may need to go to an interposition solution rather than the FUSE solution. For this reason, we decided to use library interposition in GekkoFS that exposes a distributed filesystem in top of devices.

3.3.2 Distributed FS – GekkoFS

In collaboration with Johannes Gutenberg University Mainz (JGU), we are using their development named GekkoFS (4). GekkoFS provides an adhoc filesystem but without a POSIX interface. NEXTGenIO is adding the required POSIX interfaces to run the applications selected. Our aim is to create a collaborative burst buffer using all the NVRAM of the computing nodes that are being selected by SLURM to execute a job. We will add also some API, like Export and Import files from the parallel filesystem directly from the application or other daemons (like SLURM or the datascheduler).

3.3.2.1 Data Distribution

The data of GekkoFS is automatically distributed block-wise to all the nodes in a round-robin fashion. The block size is configurable and constant across the machine. This minimises the need to keep and ask about the location of a block.

3.3.2.2 Technology

GekkoFS works as an interposition library rather than using FUSE (with all the problems that we detected once we used faster devices). We cannot do the same in echofs as we may need to develop a shared or distributed (among processes) metadata database, for this reason we decided to adopt GekkoFS as a solution. An interposition library also poses some drawbacks, as some functions may be not captured by the library and then the application will behave in an incorrect way. On the other hand, we need also to include things that FUSE does for us, like path resolution. As for example, first versions need to be used with absolute instead of relative paths.

3.3.2.3 Benchmarks – Testing

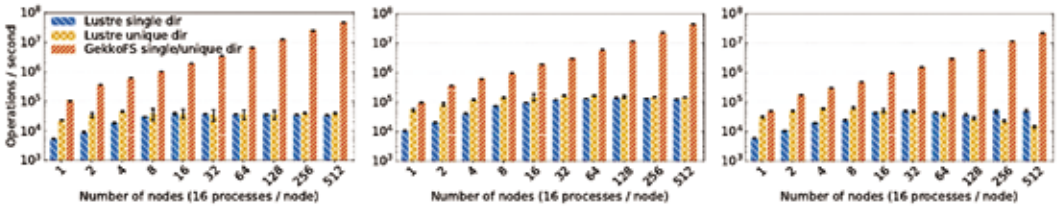
We had tested GekkoFS on the Mogon II supercomputer of JGU. The extended content of this test, among others, is described in a paper on GekkoFS accepted to the Cluster 2018 conference (4).

We simulated common metadata intensive HPC workloads using the unmodified mdtest microbenchmark and compared GekkoFS' metadata performance against the Lustre parallel file system. In our experiments, mdtest performs create, stat, and remove operations in parallel in a single directory – an important workload in many HPC applications and among the most difficult workloads for a general-purpose PFS.

Each operation on GekkoFS was performed using 100,000 zero-byte files per process (16 processes per node). From the user application perspective, all created files are stored within a single directory. However, due to GekkoFS' internally kept flat namespace, there is conceptually no difference in which directory files are created. This contrasts with traditional PFS that may perform better if the workload is distributed among many directories instead of in a single directory. Figure 11 compares GekkoFS with Lustre in three scenarios with up to 512 nodes: file creation, file stat, and file removal. The y-axis depicts the corresponding operations per second that were achieved for a particular workload on logarithmic scale. Each experiment was run at least five times and we present the mean as result.

GekkoFS' workload scaled with 100,000 files per process, while Lustre's workload was fixed to four million files for all experiments. We fixed the number of files for Lustre's metadata experiments because Lustre was otherwise detecting changing nodes when scaling to too many files.

Lustre experiments were run in two configurations: All processes operated in a single directory (single dir) or each process worked in its own directory (unique dir). Moreover, Lustre's metadata performance was evaluated while the system was accessible by other applications as well. As seen in Figure 11, GekkoFS outperforms Lustre by a large margin in all scenarios and shows close to linear scaling, regardless of whether Lustre processes operated in a single or in an isolated directory. Compared to Lustre, GekkoFS achieved around 46 million creates/s (~1,405x), 44 million stats/s (~359x), and 22 million removes/s (~453x) at 512 nodes. The standard deviation was less than 3.5% which was computed as the percentage of the mean.



(a) Create throughput (b) Stat throughput (c) Remove throughput

Figure 11: GekkoFS Metadata performance.

3.4 Object Level Storage

3.4.1 Distributed FS - dataClay

New technology NVRAM can be used, as we have already seen, to improve FS technology by significantly improving its performance. This approach helps already existing applications to benefit of such faster devices, but prevents them from taking full advantage of their performance potential. NVRAM offers byte addressable access and a speed that makes it an option to be used as memory directly used by the application, and keeps the data persistent.

In order to test these new opportunities, we have ported dataClay to use the NEXTGenIO infrastructure.

3.4.1.1 What is dataClay and How Can Applications Benefit from It?

dataClay is a next generation object store where applications can store objects as they see them in their address space. The idea is that applications implemented using an object-oriented programming language (currently Java or Python) create their complex data structures and when they want to make these data structures persistent, they just call a method “obj.MakePersistent”, and the object (i.e. a graph) will become persistent. From this point, any modification to the graph will also be persistent. In addition, when the object becomes persistent, the user/application can give it a name, and then another application can instantiate the same object from its own address space just by instantiating it using the same name.

From the point of view of applications, this mechanism offers several advantages. Firstly, persistent data becomes a first-class citizen and applications can iterate/operate over it regardless of its persistence. Persistent storage will act as a new level in the memory hierarchy and applications will not know where data is located in the same way as today applications have no idea whether a piece of data resides in the L1 cache, the LLC or in DRAM. Secondly, data will be easily shareable between applications given that only the name of an object has to be passed from one application to another, much like a file name. And thirdly, the effort in time and code needed to transform data from the memory model to the persistent model is significantly reduced – current estimations are around 30% of runtime and code size.

As part of NEXTGenIO we have modified the way data is stored in dataClay by moving from a disk and block-based kind of method to a NVRAM byte addressable backend, which suits much better the objective and architecture of dataClay.

3.4.1.2 dataClay General Architecture

The architecture of dataClay is composed of two main components: the Logic Module and the Data Service. The first one is a central repository that handles object metadata and management information. The second one is a distributed set of object stores in charge of handling the persistence of objects and executing code associated with these objects. A schematic representation of this architecture is presented in Figure 12.

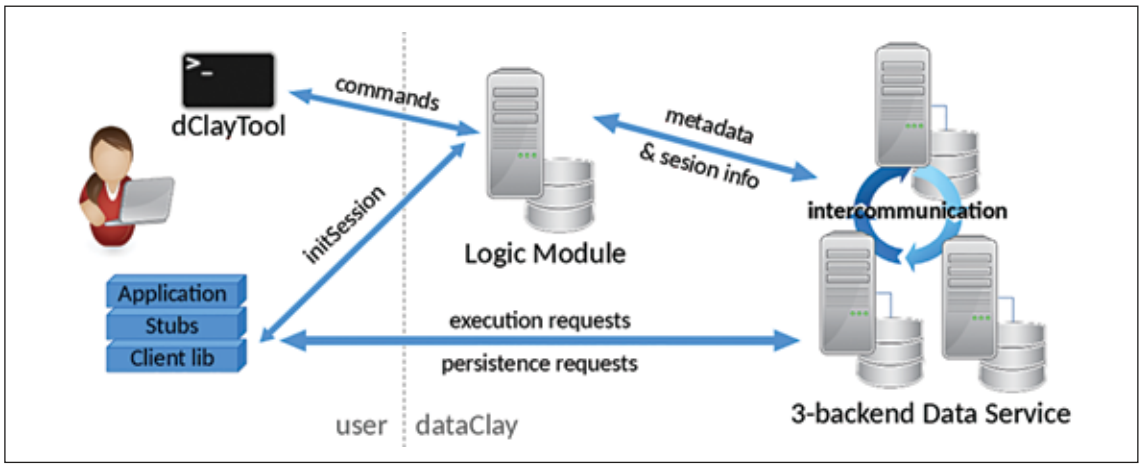


Figure 12: dataClay architecture

The **Logic Module** is a unique centralised component that keeps track of every object's metadata such as its unique identifier and location of the replicas an object may have, among other internal information. Though this service is currently centralised, it does not become a bottleneck given that it is aggressively cached in all data services and thus most of the actions are performed in a distributed manner. This aggressive caching is possible because all metadata stored in the Logic Module is either immutable (like an object ID) or can be detected as obsolete very easily (i.e. if a location of a replica is wrong, we will detect it when trying to access it).

Furthermore, the Logic Module is the entry point for any user application, which must authenticate in the system to create a working session and gain permission to interact with the components of the Data Service. Once permissions have been checked and the object has been located, all further interaction between the application and dataClay is done directly with the Data Service.

The **Data Service** is deployed as a set of multiple backends. Any of these backends handles a subset of objects as well as the execution requests directed to them (mainly the methods defined in the class of the object). This means that every backend has an Execution Environment for all supported languages (currently Java and Python) and an associated Storage System to handle object persistence.

To enable every backend to handle any kind of object and execution request, the Logic Module is in charge to deploy registered classes to every Data Service backend. In this way, every backend can load stored objects as class instances and execute class methods on them corresponding to the upcoming execution requests.

3.4.1.3 Persisting data in NEXTGenIO

As part of NEXTGenIO, we have modified the backends (main Data Service component) to use NVRAM as opposed to the disk-based mechanisms we have been using so far. Given that the performance behaviours and implications was an unknown to us at the start, we implemented three different versions that will enable us to better understand how to use NVRAM devices to store data and perform computation on this data. Having a variety of backend implementations will also ensure good support future NVRAM devices that are different from the 3D XPoint based NVDIMMs selected for the NEXTGenIO prototype, which will likely have different performance characteristics.

Key-value store on NVRAM

The mechanism that is closest to the initial version of dataClay backends is to keep persistent objects in a key-value store. For this reason, the first approach we have implemented for NEXTGenIO consists in using the NVRAM as a key-value store where we store serialised objects. Using this mechanism, we are not taking full advantage of the NVRAM hardware, but applications will benefit from a significant improvement in performance compared to the disk-based approach.

Using NVRAM as standard memory

This approach is intended to evaluate the benefits of fully integrating NVRAM into the memory hierarchy and execute code directly over the NVRAM. We are aware that this mechanism will be slower that using standard memory and assumes NVRAM will support direct access by load store operations, but on the other hand, we can avoid having to serialise/deserialise objects.

This approach is mainly implemented by mapping the non-volatile memory into the backend heap, and thus all objects instantiated will use this memory. The main objective of this approach is to understand the pros and cos of the technology for direct execution.

Using the caching mechanism of the 2LM Platform Mode

The third approach is to use the caching mechanisms of NVDIMMs where the memory is divided into two parts, one volatile and one persistent, and the device itself moves data from one space to the other as if the volatile was a cache of the persistent.

The modifications needed to implement this mechanism are equivalent to the previous one. Should this functionality not be fully available in the project prototype, we can emulate it by combining the first and second approach where NVRAM is used mainly as a key-value store.

3.4.1.4 dataClay API

Although the complete API is fully detailed in the manual[7], the main extensions to the object classes are presented here:

dataClay Object methods:

- **make_persistent**(self, alias=None, backend_id=None, recursive=True) - Stores an object in dataClay and assigns an OID to it
- **get_all_locations**(self) - Retrieves all locations where the object is persisted/replicated.
- **get_location**(self) - Retrieves the location of the object. If the object is replicated, one random location is returned.
- **new_replica**(self, backend_id=None, recursive=True) - Creates a replica of the current object.

dataClay class methods:

- **delete alias** (class, alias) - Removes the alias linked to an object. If this object is not referenced starting from a root object, the garbage collector will remove it from the system.
- **get_by_alias** (class, alias) - Retrieves an object instance of the current class (from the method is being called), that references the corresponding persistent object with the alias provided.

To demonstrate how this work, the following code implements a word count algorithm using dataClay and COMPSs.

```
#!/usr/bin/env python
import sys

from dataclay import api, getRuntime

api.init()

from WordCount.classes import TextCollection, TextStats

def count_words(text_collection):
    from pycompss.api.api import compss_barrier

    result = TextStats(dict())
    result.make_persistent()

    texts = text_collection.texts
    total_n_texts = len(texts)

    print "Ready to count words from %d different texts" % total_n_texts

    for text in texts:
        partialResult = text.word_count()
        result.merge_word_counts(partialResult)

    # Wait for result to end its reduction
    compss_barrier()

    return result

if __name__ == "__main__":
    if len(sys.argv) > 2:
        print """
    Usage:

        ./wordcount.py [text_collection_alias]
    """
        exit(1)
    elif len(sys.argv) == 2:
        alias = sys.argv[1]
        alias = "Words"

    print " # Start WordCount application #"

    words = TextCollection.get_by_alias("Words")

    res = count_words(words)

    print " # WordCount finished"
    print " # Top 10:\n %s" % res.top_words(10)
```

More examples can be found in the dataClay manual[8] and in the dataClay repository.

[7] https://www.bsc.es/research-and-development/software-and-apps/software-list/dataclay/documentation

[8] https://www.bsc.es/research-and-development/software-and-apps/software-list/dataclay

3.4.1.5 Integration with COMPSs

Given that dataClay is able to distribute objects among the different nodes in a cluster, it seems fit that they area accessed by applications running in parallel. Although the API is ready for any execution runtime or application to use it, we have made a specific effort to integrate with COMPSs (the parallel programming model at BSC) to make both better together. This integration includes an extension of the API (only designed to be used by COMPSs) to ease some of the tasks the COMPSs runtime needs to do.

The idea is that COMPSs, to reduce the data dependencies, and thus increase parallelism, creates versions of the data when it has to be modified much in the same way of the register renaming done at the architectural level. This allows the previous readers to see the old version and the new ones to see the modified one.

To support this feature, dataClay has extended its API with a **new_version** and **consolidate_version** methods that perform exactly this data renaming.

3.4.1.6 Benchmarking and testing

We had been testing dataClay on a computer with a 375 GB Optane SSD unit, and a NAND flash based NVMe SSD (SSD DC P3700).

dataClay is running in Python 2.7.15, Java 8 (1.8.0_121) and Postgres 9.5.12 default configuration. Postgres is storing the data on the Optane partition or the NAND SSD partition. We also compare some of the configurations without persistence, using memory. However, only experiments that can fit in the memory are presented.

The test does a wordcount on several files (64 files) of different sizes (32, 128 and 512 MB), it includes a phase of preparation and persistence to the device, and then an execution phase (which performs the wordcount).

On most of the scenarios, specially the bigger ones, the Optane unit surpasses the performance of the SSD by a margin of 10%.

For 32 MB and 128 MB files we have a baseline using only memory, and without persistence, of 36 seconds and 146 seconds respectively, as we can observe in Figure 13. Executions with persistence will be slower, but provides the ability to run bigger datasets and scale.

In our case, running 128 MB files on an SSD the execution time is 191 seconds and with Optane we go down to 183 seconds. Finally 32 MB files, obtain in mean a similar time on Optane and the SSD, around 45 seconds.

The time to persist the data to the device is also similar but we observed that with asynchronous persistence, Optane gets worse results with datasets below 512MB. With synchronous persistence, a 10% of benefit is observed in Optane. On the other hand, the SSD unit installed on the computer is very fast, so no big differences should be observed.

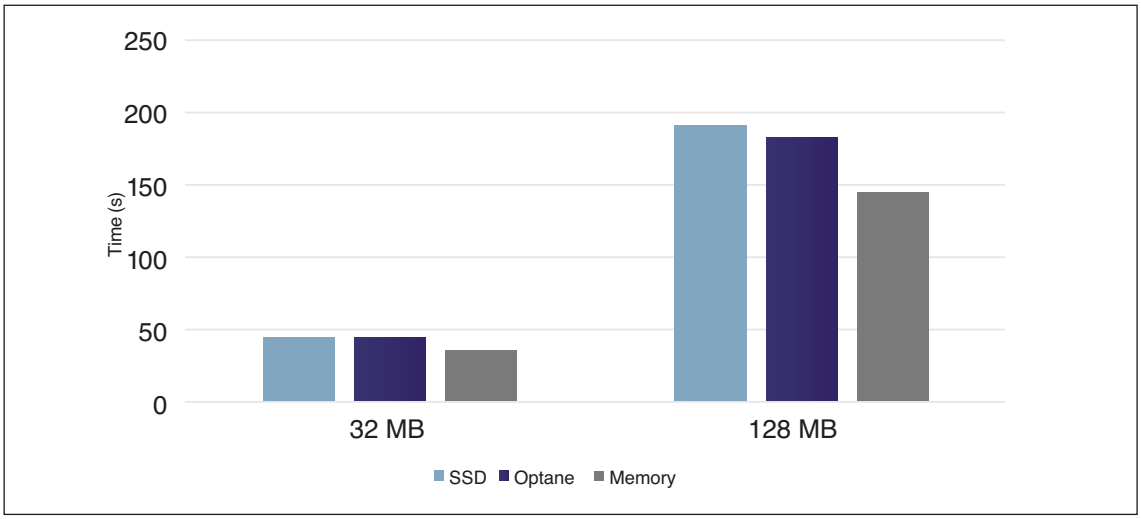


Figure 13: Wordcount dataClay performance on SSD, Optane and memory

4 Conclusions

Byte Adressable Persistent Memory with huge capacity such as Intel Optane DC Persistent Memory will be a key technology to advance storage design, from both the hardware and software perspective. In this White Paper we explored NVRAM from the software perspective showing the systemware view. Since Byte Adressable Persistent Memory with huge capacity such as Intel Optane DC Persistent Memory is a recent technology very different from traditional storage devices (including SSDs) and that latencies will be near DRAM level, we need to provide applications with different modes to test, evaluate and research which storage paradigm is the most beneficial for an application. Specialised architectures can then be implemented to have more performance for specific applications (for example, for weather research and prediction).

We finished the implementation of echofs, an ephemeral filesystem that can use different storage devices (from memory to traditional HDD, going through character devices). However, using FUSE (one of the design decisions) has introduced too many overheads. Although we have papers that show that the introduced overhead is reasonable for slower devices, it becomes problematic for faster ones. We tried to optimise the code, including changes in the FUSE libraries, however, getting a significant upgrade on the echofs performance requires removal of FUSE and implementation of an interposition library solution. One of the major concerns, and something we did not know at the start of the project is the ability to mount any filesystem (at kernel level) into the new devices, so getting better performance than a traditional filesystem in the kernel in terms of data transfer is difficult. On the other hand, we can be better on metadata operations, but this is not enough to continue supporting echofs.

Finally, using an interposition library introduces also the need to share the status between different processes, for this reason, we adopted GekkoFS and started a collaboration with JGU university GekkoFS uses an interposition library and a shared state among different instances using a distributed database. It appears to the target application as a unified space using the selected devices from the nodes. However, as a NEXTGenIO task, we leveraged the solution to include support for POSIX applications (path resolution, directories, …).

In addition to the different filesystems that are being developed or modified for NEXTGenIO we have dataClay. At the time of this White Paper (September 2018), we have a final implementation of dataClay working with PMDK libraries with different behaviours. We tested it with traditional Optane units, but more work needs to be done to analyse the behaviour in the NEXTGenIO prototype. For example, is interesting to evaluate the cost of byte-access storage to use and modify data structures directly.

5 References

1. echofs: A Scheduler-guided Temporary Filesystem to leverage Node-local NVMs. **Alberto Miranda, Ramon Nou, Toni Cortes.** 2018. SBAC-PAD.

2. **(OpenSource), PMDK.** Persistent Memory Development Kit. [En línea] http://pmem.io/.

3. **Intel and Micron.** Intel and Micron Produce Breakthrough Memory Technology. [Online] 2015. http://newsroom.intel.com/community/intel newsroom/blog/2015/intel-and-micron-produce-breakthrough-memory-technology.

4. GekkoFS – A temporary distributed file system for HPC applications. **Vef, Marc-Andre, et al.** 2018. Cluster.