nextgenio

NEXTGENIO SYSTEMWARE

NEXTGenIO Systemware White Paper

Authors: Ramon Nou, Alberto Miranda (BSC), Iakovos Panourgias, Adrian Jackson (EPCC)

16th September 2019

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

Foreword 3

About NEXTGenIO 3

1 Executive Summary 4

2 Introduction 5
2.1 Glossary 5

3 Evolution of the NEXTGenIO systemware 6

4 Scheduler 8
4.1 Slurm 8
4.1.1 Node configuration 8
4.1.2 Memory access modes 9
4.1.3 Storage configuration 10
4.1.4 Scheduling modes 10
4.1.5 Energy-aware scheduler 10
4.1.6 Metrics and Automatic Guidance 11
4.1.7 Workflows 11
4.2 NORNS 12
4.2.1 Automatic usage 12
4.2.2 Administrative usage 16
4.2.3 NORNS performance 17
4.2.4 Performance benefits for data-driven workflows 19

5 Filesystems Developed and Adapted for the Prototype 22
5.1 Block Level Storage 22
5.1.1 Distributed FS – GekkoFS 22
5.2 Object Level Storage 24
5.2.1 What is dataClay and how can applications benefit from it? 24
5.2.2 PyCOMPSs Integration 24

6 Summary 25

7 References 26

Foreword

BY **DR MICHÈLE WEILAND, NEXTGENIO PROJECT MANAGER**

NEXTGenIO has worked on improving I/O performance for HPC and data analytics workloads. The project built a prototype hardware system with byte-addressable persistent memory on the compute nodes, as well as having developed 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 developments, in particular the job and data scheduler, as well as the filesystem storage solutions that are offered on the prototype.

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 is bridging the gap between memory and storage using Intel's transformational new Optane DC Persistent Memory Modules, which will sit between conventional memory and disk storage. NEXTGenIO has designed the hardware and software to exploit the new memory technology. The goal was to build a system with 100x faster I/O than current HPC systems, a significant step towards Exascale computation.

1 Executive Summary

This White Paper summarises NEXTGenIO's work on creating and adapting schedulers (Slurm and NORNS), filesystems (GekkoFS and dataClay) and programming models (PyCOMPSs) to the NEXTGenIO architecture. All the software has been released and is available on the prototype.

In this document, we present an overview of the different components that have been developed, together with evaluations, conclusions and lessons learned. The White Paper describes the new functionality that was developed for Slurm, in particular around the use of NVRAM and workflows. The scheduler has been enhanced to be able to take advantage of the persistent memory (for example by scheduling related jobs on the same nodes).

The White Paper also introduces the NORNS data scheduler, designed to manage data transfers asynchronously between different backends. NORNS is fully integrated with SLURM.

Finally, we present the ad-hoc filesystem GekkoFS, which allows the creation of a distributed filesystem from the local space of each computing node, and the dataClay object store and its integration with PyCOMPSs. Thanks to this integration, it is possible for both Python and Java workloads to take advantage of NVRAM.

2 Introduction

NVRAM provided by Intel DCPMM is the technology that makes NEXTGenIO different from other architectures. As it is a new architecture, supporting different storage paradigms (normal filesystems, GekkoFS, Object Stores etc.) and modes (1LM, 2LM or mixed) is necessary in order to perform research and find out which options are best suited for an application. All these modes need to be supported by the capabilities of the Job Scheduler (**Slurm**), which is the entry point for the user. As storage is a key point in NEXTGenIO, we also propose the concept of a Data Scheduler and Data Mover with **NORNS**.

2.1 Glossary of Acronyms

Acronym	Description
1LM	1LM has 1 level of memory. This represents the mode where DRAM and NVRAM are used as separate memory structures, with their own memory address space. 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 provides two memory levels. This represents the mode where the DRAM is used as a transparent 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™	3D XPoint™ technology is a simple, stackable, and transistor-less memory design that creates fast, inexpensive, and non-volatile storage memory with low latency.
API	Application programming interface
COMPSs	COMP Superscalar: Programming model for distributed infrastructures
DCPMM	Data Centre Persistent Memory Module
DFS	Distributed File System
DRAM	A random-access memory (RAM) that uses a single transistor-capacitor pair for each bit
E.T.A.	Estimated Time to Arrival
GekkoFS	Adhoc Filesystem from University of Mainz and BSC (Barcelona Supercomputing Center) partially developed in the NextGenIO project
MAP	Performance monitoring tool for parallel, multithreaded or single threaded C++, Fortran codes
Memory-Side Cache	A memory-side cache holds blocks recently requested from memory. It may be viewed as another level of the memory hierarchy. It is associated logically with a corresponding general-purpose memory and is therefore coherent
mmap	Memory mapped. Method for memory mapped file IO
MPI	Message Passing Interface. A standard for message-based distributed parallel computing
NVM	Non-volatile memory is capable of retrieving stored data even after a power outage. Examples of non-volatile memory include 3D NAND, solid-state drives, and 3D XPoint™ technology
NVRAM	Non-volatile memory (see above)
PFS	Parallel File System
PMDK	Persistent Memory Development Kit
POSIX	Portable Operating System Interface. Standard specified by IEEE for maintaining compatibility between the operating systems
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
PyCOMPSs	Python binding of COMPSs
RPC	Remote Procedure Call, a client calls a function on a server. This can be in the local node or in another node
SSD	Solid State Drive
Syscall	Function implemented in the kernel, that is called through an standard interface. i.e., open, write, read

3 Evolution of the NEXTGenIO systemware

The key aim that was driving the systemware development in the NEXTGenIO was to offer a fully configurable environment for the user, controlled by the Job Scheduler (Slurm). The rationale behind this choice was that NEXTGenIO introduces new hardware that has many different usage scenarios and configuration options. If application A is different from application B, the available configurations and options should be easily selectable from the job manager. Instead of having a fixed partition of compute nodes in a certain configuration, it is possible to request for the job scheduler to set up compute nodes in a specific configuration. Slurm will reboot and configure such nodes as needed. All required steps are integrated within the normal workflow.

As rebooting a node and reconfiguring it is not a quick task, making large changes on the system may not be efficient. However, we expect that only some jobs will need such reconfiguration. Furthermore, jobs are typically submitted in groups, so the reconfiguration overheads should be minimal overall.

In the software layer, particularly in relation to the filesystems, we expect similarly diverse user requirements: if application **A** uses a lot of metadata, application **B** uses objects and application **C** is mainly streaming data, the filesystem requirements of each application will be different. With this scenario in mind, we also include filesystem reconfigurability in **Slurm**.

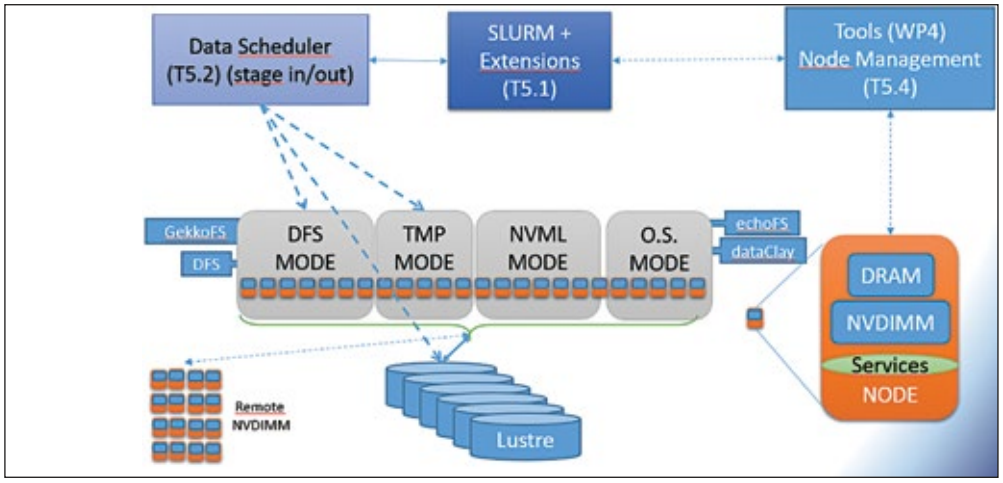

Figure 1: System Diagram. Overview of the different systemware components.

We also considered legacy applications and how they can work with the Intel® OptaneTM Data Center Persistent Memory Module (DCPMM) implementation of NVM. The aim was to create a new filesystem, transparent to the application, which could work with any memory (or storage) in the compute node. We joined efforts with the **GekkoFS** (1) developers from University of Mainz with the aim of adding POSIX compliance on top of this distributed file system. **GekkoFS** is a library interposition filesystem and it was created with a distributed architecture in mind. The collaboration with University of Mainz has been a notable success: we can support distributed applications and the GekkoFS filesystem option is fully integrated into Slurm. Additionally, Intel have released a syscall interception system based on binary modification, which addressed a number of initial issues with the GekkoFS library interception system.

The other storage system adapted for NEXTGenIO is **dataClay**, an object store with a special capability: it can execute and store methods. It was created with NVM in mind and is able to do byte-level changes instead of block-level. This made inclusion of **dataClay** in the software to be tested on the system a logical choice. As the hardware was not ready at the time of development, we created several versions of the **dataClay** backends to NVM and tested them on the final prototype hardware.

We are not limited to using the filesystems described above. Using a similar approach we can include other filesystems such as **DAOS** (2) or even a downsized **Lustre**.

For all filesystem options, we need a way to transfer data into and out of the node efficiently. For that reason we developed **NORNS**, which can decide when and how to transfer data. NORNS is able to transfer data from/to different backends asynchronously and has a wide range of options. We created various scheduling algorithms inside Slurm to support all possible configurations.

The final systemware component is **PyCOMPSs**, a programming model that enables us to run parallel Python software on the prototype along with AI software such as Tiramisu (which is based on TensorFlow). **PyCOMPSs** is integrated with **dataClay** and the monitoring/debugging tools from WP4. **GekkoFS** is currently not compatible with **PyCOMPSs** due to some issues with the interception system.

4 Scheduler

All HPC systems use a Job Scheduler. On NEXTGenIO we chose the Slurm scheduler (3) because it is open source and can be extended with new functionalities. We have modified Slurm in different places to include extended support for NEXTGenIO.

4.1 Slurm

4.1.1 Node configuration

Slurm enables the (re-)configuration of the NEXTGenIO compute nodes. As depicted in Figure 2, the hardware supports different memory access modes: the 1LM mode where DRAM is directly accessible and the 2LM mode where DRAM acts as cache for the persistent memory space. The DCPMMs can be partitioned to reserve space for Memory mode access and App Direct access. While App Direct space can be used in both platform modes (1LM and 2LM), the Memory mode partition can only be used in 2LM mode.

2LM supports large volatile main memory with up to 6TB in a 2-socket system. In 2LM mode, the DRAM cache hides the additional latency of the persistent memory. No application modification is required.

The 1LM mode supports access to fast DRAM and to persistent memory via App Direct access. To take full advantage of App Direct access, applications need to be customised, or alternatively they can also use a filesystem as described in Section 4.1.3 that uses the advantages of App Direct instead. In this case no application modification is required. To switch between modes a reboot is required.

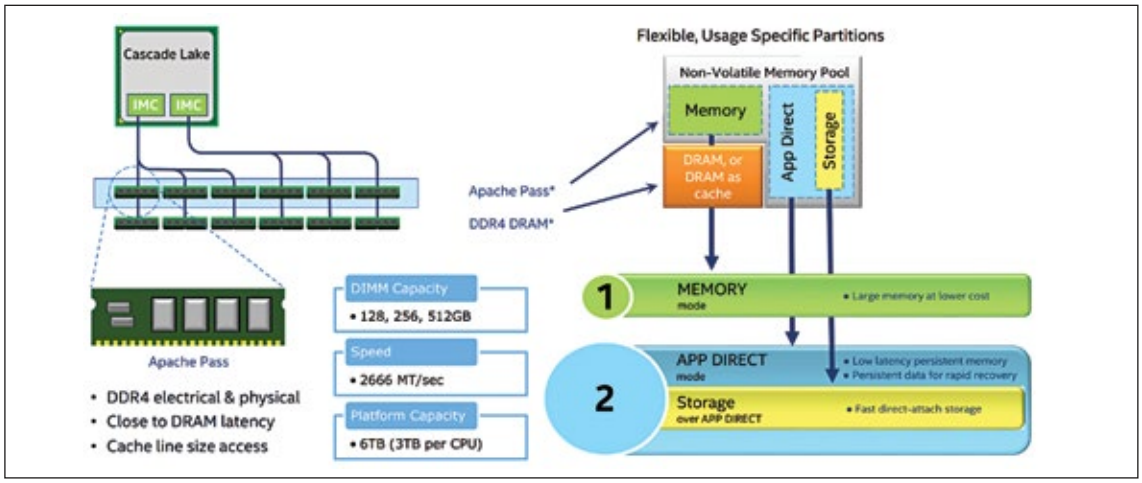

Figure 2: DCPMM usage models.

We added a set of operations to the core functionality of the scheduler to manage both the memory platform modes and the required configurations on the DCPMMs. Slurm allows the user to query, create, delete, modify, and assign NVRAM namespaces, set the interleave set and size and swap from 1LM to 2LM mode.

The different modes, which are described in the next subsections, are selectable by Slurm with these command line options:

* `--nvram-options=1LM:1000`
* `--nvram-options=2LM:1000`

4.1.2 Memory access modes

4.1.2.1 Memory mode access

The DCPMMs can be configured such that a portion or all of the capacity is allocated as a volatile memory partition in what is called Memory mode. In this case the entire DRAM capacity is used as cache in front of this memory mode partition on the DCPMMs. In other words, the volatile main memory address space seen by the processor is as large as the Memory mode partitions of all allocated DCPMMs combined. DRAM cannot be accessed explicitly. This organisation is depicted in Figure 2 by the green ("Memory") and orange ("DRAM Cache") blocks. This platform mode is called 2LM, for two-level memory. It reflects the organisation of volatile memory in two levels, namely the DCPMMs and the DRAM modules as cache. Note that the remaining DCPMM capacity can still be allocated as persistent partitions for App Direct access.

The DCPMM's Memory mode partition behaves as volatile memory despite the persistent nature of the media. This is because modified data can be in the volatile copy back DRAM cache and there is no guarantee that there is always a valid copy in the DCPMMs. As a result, data stored in the DCPMM partition in this mode is discarded according to standard POSIX rules (4) and is lost upon power and system failures. This is achieved through encryption and key management in hardware such that no data erasure is needed and the stored data is secure. Memory management in this mode can be done through any existing volatile memory allocator, such as the popular glibc's malloc (5) and jemalloc (6).

The Memory mode configuration provides applications with transparent access to large memory capacities without the need for modifications. Thus, it is best suited to applications that do not require persistence of data and benefit from large memory. On the other hand, because DCPMM latencies are higher than those of DRAM, the performance observed for applications is highly dependent on the effectiveness of the DRAM cache. Memory mode configurations are best used for applications with access patterns that exhibit good locality.

4.1.2.2 Application Direct (App Direct) mode

All or part of the DCPMM capacity can be configured in Application Direct (App Direct) mode to become part of a persistent partition. The remainder of the DCPMM capacity can be configured in any of the other modes. This organisation is depicted by the blue ("App Direct") blocks of Figure 2.

This mode exports the DCPMM App Direct space as direct accessible memory with persistent characteristics. This means that: i) direct CPU loads and stores from user space are possible (hence the "App Direct" name of this mode) and ii) data placed in the DCPMM area configured in the App Direct mode is persistent. In practice, memory configured in this mode can be managed and used in two ways. In the first way, it is used as large volatile memory. In this case, unlike in memory mode, there is no DRAM cache and application accesses go directly to the NVRAM. In this case, since the data is naturally persistent, it is up to the application to erase or encrypt data to make it secure, as well as to relinquish the memory allocation when no longer needed. Libraries, such as libmemkind (7) and libvmmalloc (8) exist to facilitate such volatile usage. In the other way, the NVRAM capacity allocated in App Direct mode is used as persistent memory, storing data for potentially long periods of time. Access is via direct loads/stores from the application in user space, which sets this apart from conventional block devices (see also 4.1.2.3 below). It is up to the application to manage this persistent storage, such as by creating a persistent name space and enforcing access control. In practice, the recommended way for applications to do this is to use a persistent memory aware file system, such as ext4 or XFS with DAX support[1]. Libraries, such as the Persistent Memory Development Kit (PMDK)[2], operate on top of a persistent memory aware file system to facilitate usage of storage-class memory.

To use this memory configuration, applications need to be modified. The volatile memory usage scenario described above is best suited to applications that do not require persistence of data and that either can cope with the longer latencies of NVRAM compared to DRAM, or can manage the staging of data through DRAM (in the 1LM platform configuration) by prefetching or implementing a software managed cache. The persistent memory usage scenario is best suited to applications that require persistence of data and that can be heavily modified to exploit the App Direct mode.

[1] https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[2] http://pmem.io/pmdk

4.1.2.3 Use as Storage Target

A portion the DCPMM capacity can be configured to serve as a storage target, being accessed as a block device. The remainder of the NVDIMM capacity and the DRAM capacity can then be configured in any of the other modes. This organisation is depicted by the yellow ("Storage") blocks in the left and right portions of Figure 2.

This mode exports the (persistent) memory to applications as a traditional storage block device. Thus, it is meant to be used in unmodified legacy applications and to implement traditional file systems. Unlike traditional block devices, however, it is possible to have direct media access without the need to stage data through a RAM page cache. However, like a traditional block device, access still must be through the OS and no direct access is possible from user space applications. For all intents and purposes, a storage partition on NVDIMM behaves like an ultra-fast SSD.

4.1.3 Storage configuration

When operating in 1LM platform mode we can select different filesystems. As it is the general preferred option, we have integrated the GekkoFS distributed filesystem into Slurm. Filesystems can be set up with the following sbatch command line arguments, and additional filesystems can be integrated easily.

```
--filesystem-device=/mnt/pmem_fsdax0/
--filesystem-type=gekkofs
--filesystem-mountpoint=/mnt/pmem_fsdax0/gkfs_mnt
```

4.1.4 Scheduling modes

Scheduling modes are created to enhance Slurm capabilities and be able to manage the new options. For example, a storage scheduling option puts emphasis on scheduling to nodes that will not need reconfiguration of the NVDIMM modes to run a job.

4.1.5 Energy-aware scheduler

We have enhanced Slurm's scheduling algorithms to take into account the current temperature of each node. Slurm will try to allocate jobs to cooler regions of a system in order to even out the temperature across the infrastructure. This functionality can be turned on/off by the administrator and it will not affect jobs that are currently running or have already been allocated (i.e. Slurm will not re-run the scheduling loop). However, subsequent jobs will be allocated based on the new policy.

There is a new option, `--optimise-for-energy`, to be included during job submission, which enables different features such as:

- Throttling down the CPU frequency (which should have minimal performance impact on memory bound applications);
- Modifying the CPU governor (which should have minimal performance impact on memory bound applications);
- Reducing power on the DCPMMs modules (which should have minimal performance impact on applications that mostly read from NVRAM);
- Increase energy optimisations for traditional HDDs (if present);
- Reduce fan RPMs (on supported hardware).

Slurm will revert these optimisations to the default state when launching subsequent jobs. However, nodes that are not allocated to jobs will stay in the reduced energy consumption state. Nodes can also be shut down, to conserve even more energy. Due to the significant boot times, however, all nodes in the NEXTGenIO prototype are kept online by default.

4.1.6 Metrics and Automatic Guidance

We can gather metrics from the NVRAM devices and those metrics are integrated in the Slurm Accounting database, where the metrics are associated with nodes, job IDs and owners.

Slurm gathers a plethora of metrics from the compute nodes. Some of the metrics are persistently stored in the Accounting database; whilst others are only kept at the runtime level.

The following metrics pertaining to NVRAM are stored:

- Availability of NVRAM;
- Total size of NVRAM;
- Number and size of namespaces.

Furthermore, Slurm collects the following general metrics:

- Temperature metrics (for applicable devices);
- Lustre filesystem metrics;
- NUMA metrics;
- CPU usage metrics;
- Memory usage metrics.

Finally, Slurm was updated to collect and store the following “real-time” metrics:

- CPUFrequency;
- CPUTime;
- CPUUtilization;
- Pages;
- Power;
- RSS;
- ReadMB;
- Reads;
- VMSize;
- WriteMB;
- Writes;
- LustreReads;
- LustreWrites.

The “real-time” metrics have a small delay (in the order of several seconds; less than a minute) and can be viewed by users and administrators.

4.1.7 Workflows

Another option added to Slurm is the capability to set up and run workflows directly with the Job Scheduler. The rationale of this is that as the hardware is reconfigurable and we have persistent memory available, different elements of a workflow should be run on the same nodes to avoid copying data from/to the parallel file system. The following sbatch command line options can be used to set up a workflow:

```
--workflow-start [The job starts a workflow]
--workflow-prior-dependency=jobid [Sets a dependency on the jobid]
--workflow-end [The job ends a workflow]
```

Each workflow has a WorkflowID, and the users can request the overall status of a workflow and get a list of all the jobs and their status. If a step in a workflow fails, all the subsequent jobs belonging to it are cancelled.

4.2 NORNS

NEXTGenIO designed and implemented NORNS, a data scheduling service which facilitates interoperation between Slurm and the different storage tiers present in the prototype. Using NORNS, users and applications can request asynchronous data transfers between compute nodes and each storage tier, regardless of whether a tier is local or remote. NORNS is a service written in C++ that interfaces with Slurm and provides the necessary APIs and facilities to submit, monitor, and manage asynchronous data transfers between the different storage tiers available in the prototype. Furthermore, it has been designed to abstract users from the technical details of the different storage tiers, be they NVM, SSDs, traditional hard disks or parallel file systems.

4.2.1 Automatic usage

Users can take advantage of NORNS automatically through Slurm job directives or by directly invoking the NORNS API in application code. The aforementioned Slurm job directives have been especially developed for the NEXTGenIO project and are useful for applications that have not been fully altered to manually exploit non-volatile memory directly, as well as for applications that wish to share data between different stages of a workflow. Through these directives, users can currently request data to be transferred between filesystems and compute nodes, between compute nodes, and between individual jobs in a workflow. This includes functionality such as:

- Staging files in and out for an application
- Keeping data on node(s) for future application(s) use
- Moving data between nodes to match future job schedules

These functionalities are supported via the Slurm directives shown in Figure 3:

```
# Stage in a resource RID from dataspace 'origin' to dataspace 'destination'
# with an optional resource mapping
#NORNS stage_in origin:RID destination:RID [mapping]

# Stage out a resource RID from dataspace 'origin' to dataspace 'destination'
# with an optional resource mapping
#NORNS stage_out origin:RID destination:RID [mapping]

# Share a resource RID at dataspace DS between workflow stages
#NORNS persist_store|delete|share|unshare DS:RID [user]
```

Figure 3: Slurm directives for NORNS integration

By using these directives in addition to the Slurm extensions for data-driven workflows described in Section 4.1.7, it is possible for users to define workflows of batch jobs where intermediate data is kept at all times in NVM, with Slurm taking care of instructing NORNS to make any data transfers required to support the execution of each step. The code excerpts shown in Figure 4 and Figure 5 demonstrate how a simple workflow with a preprocessing stage and a simulation stage can be defined with intermediate data being shared between stages using NVM.

run-preprocessing.slurm

```
#!/bin/bash

### Usual boilerplate to define a job
#SBATCH --job-name=lhc-preprocessing- 01
#SBATCH --time=04:30:00
#SBATCH -o prepro_01_out_%J
#SBATCH -e prepro_02_err_%J
#SBATCH --nodes=5

### Copy data from the 'lustre' dataspace into the 'pmem0' dataspace
### of each node enforcing the mapping provided by the user in
### ${HOME}/preprocessing-mapping.dat. The mapping file
### instructs NORNS to copy data from the 'lustre://data/lhc/raw'
### directory into 'pmem0://workflow42' (see below for the actual
### file contents).
#NORNS stage_in \
       lustre \ ### This will map to ${NORNS_ORIG_DS} in the mapping file
       pmem0  \ ### This will map to ${NORNS_DEST_DS} in the mapping file
       ${HOME}/preprocessing-mapping.dat

### Make sure that the 'workflow42' directory is made persistent after
### this workflow stage completes (otherwise Slurm would delete all the
### data in node-local storage to prepare the compute node for the next job)
#NORNS persist store pmem0://workflow42

### Actual code for running the preprocessing stage
module load impi
module load gnu8

export I_MPI_HYDRA_TOPOLIB=
export I_MPI_FALLBACK=0
export PSM2_DEVICES="self,hfi,shm"

### The ${SLURM_NODEID} variable is an environment variable propagated
### to the user processes that can be used to determine the node index in
### the allocation running the code. As such, we can use it to find out
### the input data that needs processing
srun \
    -n ${SLURM_JOB_NUM_NODES:?} \
    -N ${SLURM_JOB_NUM_NODES:?} \
    --oversubscribe \
    --export=ALL \
    filter \
      --gamma=63.7 \
      --burst=720.0+ \
      --input=${NORNS_PMEM0}/workflow42/lhc_input_${SLURM_NODEID} \
      --input=${NORNS_PMEM0}/workflow42/lhc_input_$((SLURM_NODEID+5))
```

Figure 4: Example of a workflow pre-processing phase using NORNS through Slurm

The job definition shown in Figure 5 relies on the mapping file ${HOME}/preprocessing-mapping.dat to control how input data needs to be mapped into compute nodes. For this particular example, the mapping file contains the directives shown in Figure 5.

preprocessing-mapping.dat

```
### INPUT/OUTPUT FILE MAPPINGS ###
### This definition instructs NORNS to transfer data from the 'lustre'
### dataspace the following input data distribution in 'pmem0':
###   nextgenio-cn01: workflow42/lhc_input_0.dat  workflow42/lhc_input_5.dat
###   nextgenio-cn02: workflow42/lhc_input_1.dat  workflow42/lhc_input_6.dat
###   nextgenio-cn03: workflow42/lhc_input_2.dat  workflow42/lhc_input_7.dat
###   nextgenio-cn04: workflow42/lhc_input_3.dat  workflow42/lhc_input_8.dat
###   nextgenio-cn05: workflow42/lhc_input_4.dat  workflow42/lhc_input_9.dat
['${NORNS_ORIG_DS}://data/lhc/raw/lhc_input_%[0-9]+%.dat'];
0;${NORNS_DEST_DS}://;0,5
1;${NORNS_DEST_DS}://;1,6
2;${NORNS_DEST_DS}://;2,7
3;${NORNS_DEST_DS}://;3,8
4;${NORNS_DEST_DS}://;4,9
```

Figure 5: Example of a mapping file to control input data distribution in compute nodes

The second stage of this sample workflow simply runs a simulation using the data generated in NVM by the preprocessing stage and instructs Slurm and NORNS to stage out the simulation results to the Lustre parallel file system. Code which effects this is shown in Figure 6.

run-simulation.slurm

```
#!/bin/bash

### Usual boilerplate to define a job
#SBATCH --job-name=lhc-preprocessing- 01
#SBATCH --time=04:30:00
#SBATCH -o prepro_01_out_%J
#SBATCH -e prepro_02_err_%J
#SBATCH --nodes=5

### Copy data from the from the 'lustre' dataspace
### into the 'pmem0' dataspace of each node following the mapping provided by
### the user in ${HOME}/preprocessing-mapping.dat. The mapping file
### instructs NORNS to copy data from the 'lustre://data/lhc/raw'
### directory into 'pmem0://workflow42' (see below for the actual
### file contents).
#NORNS stage_out \
       pmem0  \ ### This will map to ${NORNS_ORIG_DS} in the mapping file
       lustre \ ### This will map to ${NORNS_DEST_DS} in the mapping file
       ${HOME}/simulation-mapping.dat

### Make sure that the 'workflow42' directory is deleted after the job
### completes, since data is no longer required for this worklow
#NORNS persist delete pmem0://workflow42

### Actual code for running the preprocessing stage
module load impi
module load gnu8

export I_MPI_HYDRA_TOPOLIB=
export I_MPI_FALLBACK=0
export PSM2_DEVICES="self,hfi,shm"
export PSM2_MULTIRAIL=1
export PSM2_MULTIRAIL_MAP=0:1,1:1

srun \
    -n ${SLURM_JOB_NUM_NODES:?} \
    -N ${SLURM_JOB_NUM_NODES:?} \
    --oversubscribe \
    --export=ALL \
    /home/software/lhc_simulation \
      --steps=5000 \
      --input=${NORNS_PMEM0}/workflow42
```

Figure 6: Example of a workflow simulation phase using NORNS through Slurm

With these scripts in place, the workflow can then be scheduled to run with the commands shown in Figure 7:

```
$ sbatch --nodes=5 \
         --workflow-start \
         run-preprocessing.slurm
Job ID: 5656

$ sbatch --nodes=5 \
         --workflow-prior-dependency=5656 \
         --workflow-end \
         run-simulation.slurm
Job ID: 5666
```

Figure 7: Example of a workflow submission

4.2.2 Administrative usage

Given that the ability to send commands to the NORNS transfer daemons directly from the command line is often useful for system administrators, NORNS provides a command line interface called norns-control that allows interactive communication with the daemons. The code samples in Figure 8 demonstrate several example commands.

```
# List available commands
(NORNSControl) help

Documented commands (type help <topic>):
========================================
dataspace ping task

# Check if the transfer daemon is running
(NORNSControl) ping
urd daemon at "nextgenio-cn01" is UP

# Create a DAXFS dataspace with id "nvml0" and 10GiB of quota
(NORNSControl) dataspace create nvml0 daxfs /mnt/pmem_fsdax0 10000

# Submit an administrative copy task between existing dataspaces
(NORNSControl) task submit copy nvml0:data.bin lustre:data.bin
Task 42 submitted
(NORNSControl) task submit copy nvml0:data.bin nvml0@nextgenio-cn02:data.bin
Task 43 submitted

# Check task status
(NORNSControl) task status 42,43
Task 42 finished successfully
Task 43 finished with error: "No such file or directory"

# Remove an existing dataspace
(NORNSControl) dataspace remove nvml0
```

Figure 8: Administrative commands for transfer daemons

4.2.3 NORNS performance

As an infrastructure service for job schedulers and applications, NORNS must add as little overhead as possible to the normal operations of its intended clients, and it is also important for NORNS to transfer data efficiently, so that jobs are not delayed due to staging operations. In this section, we evaluate how NORNS performs in the NEXTGenIO prototype when serving concurrent requests from multiple clients. We also evaluate its data transfer rates when moving data between compute nodes.

4.2.3.1 Request rate

Figure 9 and Figure 10 show, respectively, the performance of the NORNS service when serving requests (128 bytes) from local processes and from other NORNS instances in remote compute nodes. For local requests, we create up to 32 concurrent processes that submit $50{\times}10^3$ consecutive requests to the local transfer daemon using the NORNS API. For remote requests, we use up to 32 of the 34 compute nodes to send $50{\times}10^3$ remote requests in parallel to the same NORNS target instance, both sequentially and in groups of 16. We configure NORNS to use the ofi+tcp plugin since it is less performant than other fabric-specific plugins and thus provides a lower bound for performance. In both cases we measure the latency taken by the daemon to respond to the client (i.e. the time taken to process the request, create a task descriptor, add it to the task queue, and respond to the client) as well as the observed aggregated throughput in requests per second (RPS). Each experiment is repeated 20 times.

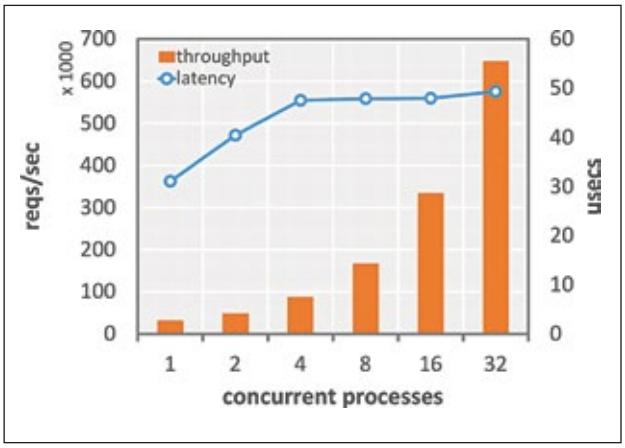

Figure 9: NORNS throughput and latency serving local requests

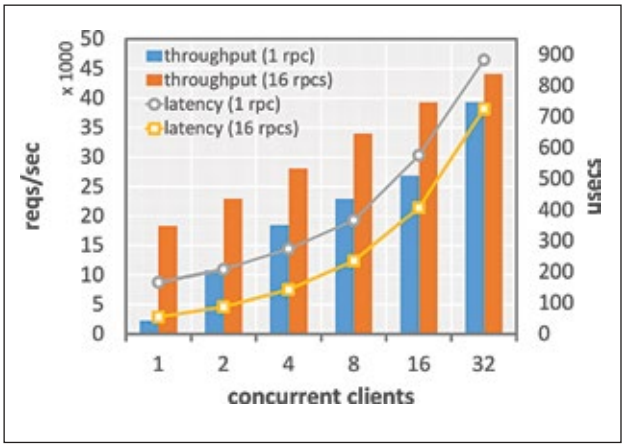

Figure 10: NORNS throughput and latency serving remote requests

The results show that NORNS exhibits low latency when serving local requests (≈50 microseconds in the worst case), which is expectedly higher for remote requests (≈900 microseconds in the worst case) since they involve network communication rather than node-local inter-process communication. Throughput scales up to ≈700,000 local RPS and up to ≈45,000 remote RPS, which should be enough to support the expected load from the Slurm daemon and applications.

4.2.3.2 Data Transfer Rates

Figure 11 and Figure 12 show, respectively, the data transfer rates when reading/writing from/to remote NORNS transfer daemons. The benchmark measures the aggregated bandwidth from up to 32 clients reading/writing data in parallel from a single NORNS target. Again, we measure the performance using the ofi+tcp plugin and show results with 1 Remote Procedure Call (RPC) and 16 RPCs in flight. NORNS clients use a 16 MiB buffer for transfers since increasing the buffer did not improve bandwidth further, and each experiment is repeated 20 times.

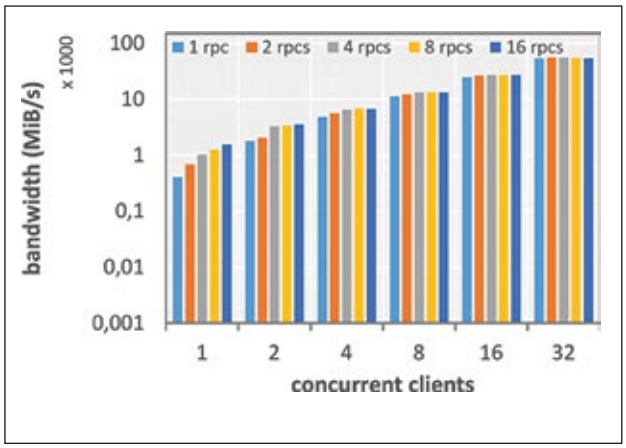

Figure 11: NORNS aggregated bandwidth for remote data reads

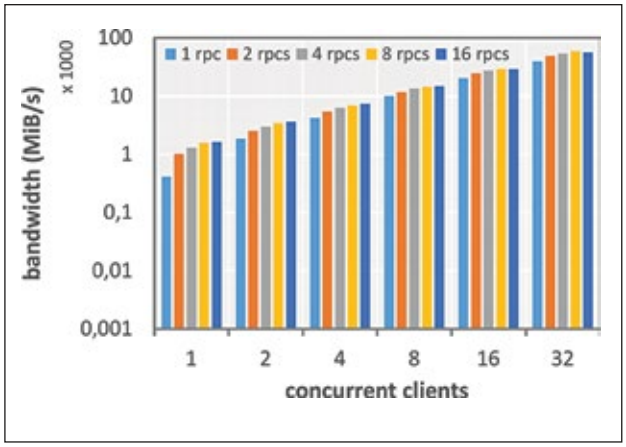

Figure 12: NORNS aggregated bandwidth for remote data writes

The results show that aggregated bandwidth scales linearly with the number of nodes peaking at ≈55.6GiB/s for reads and at ≈59.7GiB/s for writes. A detailed examination of the results shows that the bandwidth per client saturates at ≈1.7GiB/s for reads and ≈1.8GiB/s for writes, and that it remains stable (without either rising or falling) even if the number of in-flight RPCs or nodes is increased. This suggests that the Mercury-based network manager of NORNS is capable of reaching the maximum bandwidth for this particular protocol and that transfer performance is limited by the underlying network stack.

4.2.4 Performance benefits for data-driven workflows

We also evaluated the performance benefits achievable when utilising the functionality described in Section 4.1.7 to maintain data on compute node NVMs and share that data between compute jobs in a workflow. We evaluated performance both with a synthetic benchmark and with a user application (OpenFOAM), the results of which will be discussed in the following sections.

4.2.4.1 Synthetic benchmark

We created a synthetic workflow benchmark that has a producer and a consumer of data, configurable to produce a range of files with a range of different sizes. We can run this benchmark either targeting the Lustre filesystem or the NVMs on each compute node and observe the performance difference between utilising both forms of storage. We can also utilise the integration with Slurm and NORNS to maintain data on compute node NVMs between workflow phases and observe the performance benefit this provides. Table 1 outlines the performance achieved when producing and consuming 100GiB of data running the workflow on Lustre or directly on NVMs. Each benchmark workflow ran 5 times and we report the mean result. Performance varied by <5% across runs. Benchmarks were compiled using the Intel 2019 compiler with the -O3 flag.

Workflow component	Lustre	NVM
Producer	96 secs	64 secs
Consumer	74 secs	30 secs
TOTAL	**170 secs**	**94 secs**

Table 1: Synthetic workflow runtime (Lustre vs NVM)

For the benchmark targeting Lustre we ran the producer and consumer on two separate compute nodes to ensure that data I/O caching in the computing node operating did not affect the measured runtimes. For the workflow using NVMs we ran in two different configurations, one with the components re-using the same node and the data stored on the NVMs, and one using different compute nodes with data moved off the producer node after production and pre-staged on to the consumer node before the consumer is run.

For the staging benchmark we ran another application on the nodes where the data staging was occurring (both post-producer and pre-consumer staging) to evaluate the potential impact of operations NORNS and the job scheduler may undertake on the running applications. For this, we chose to run the High Performance Conjugate Gradients (HPCG) benchmark[3], a benchmark which claims to represent the computational and data access patterns of a broad set of important computational simulation applications. The conjugate gradients algorithm used in the benchmark is bound by the performance of the memory system, and to a lesser extent of the network used to connect the processors together. We ran a small HPCG test case that would complete in ≈122 seconds using 48 MPI processes per node.

We can see from Table 1, which outlines the performance of using Lustre directly, and of running the workflow components consecutively on the same node, that using local NVM storage gives ≈46% faster performance (96 vs 170 seconds) overall runtime for the workflow compared to using Lustre. Note, for the NVM case we run a job that reads and writes 200GB of data between workflow components on the same node to ensure caching does not affect performance.

[3] http://www.hpcg-benchmark.org

Component	Runtime	Impact
HPCG baseline (no staging)	122 secs	–
HPCG + stage in	142 secs	16.39% slower
HPCG + stage out	**137 secs**	**12.29% slower**

Table 2: Performance impact on HPCG due to concurrent data staging

When considering the staging of data benchmark, where data is moved to Lustre after the producer has completed and loaded in to NVM before the consumer starts, we see outlined in Table 2 that the Producer and Consumer tasks are not affected by this mode of operation. We achieve performance commensurate with running both tasks on the same node using node-local NVM. However, it is evident that the application running on the compute nodes whilst the data is being moved to and from the node-local storage is impacted by that activity. We experience an approximately 15% increase in runtime for the HPCG benchmark. It is worth noting that this is likely to be a disproportionately large impact compared to general applications as the runtime of HPCG is similar to that of the time required to stage the data to or from the NVM, and as HPCG is memory-bound. For most applications this activity would normally only impact the beginning and/or end of execution, when the job scheduler is preparing for a job finishing and the next job starting, and thus should impact a much smaller part of the application execution. However, it is important to note there is potential for impact on running applications.

4.2.4.2 OpenFOAM Performance

OpenFOAM (9) is a C++ object oriented library developed to provide Computational Fluid Dynamics functionality that can easily be extended and modified by users. It is parallelised with MPI and is heavily used in academia and industry for large scale computational simulations. OpenFOAM often requires multiple stages to complete a simulation, from preparing meshes and decomposing them for the required number of parallel processes, to running the solver and processing results. It also often undertakes large amounts of I/O, reading in input data and producing data for analysis. It is common that the different stages require differing amounts of compute resources, with some stages only able to utilise one node, and others (such as the solver) requiring a large number of nodes to complete in a reasonable amount of time. In general, OpenFOAM favours creating a directory per process that will be used for the solver calculations, necessitating a large amount of I/O for big simulation. Given these features, OpenFOAM is a strong target for both workflow functionality and improved I/O performance through node-local I/O hardware.

Stage	Lustre	NVM	Benefit
Mesh decomposition	1191 secs	1105 secs	–
Data staging	–	32 secs	–
Solver	123 secs	66 secs	46% faster
Total	**1314 secs**	**1203 secs**	**8% faster**

Table 3: Performance benefits of automatic data staging on OpenFOAM workflow

For this benchmark we ran a low-Reynolds number laminar-turbulent transition modelling simulation of the flow over the surface of an aircraft (10), using a mesh with ≈43 million mesh points. We decomposed the mesh over 16 nodes enabling 768 MPI processes to be used for the solver step (picoFOAM). The decomposition step is serial, takes 1105 seconds, and requires 30GB of memory. We ran the solver for 20 timesteps, and compared running the full workflow (decomposing the mesh and then running the solver) entirely using the Lustre filesystem with using node-local NVM with data staging between the mesh decomposition step and the solver. The solver produces 160GB of output data when run in this configuration, with a directory per process. Running the solver using Lustre required 123 seconds, whereas running the solver using node-local storage required 66 seconds, close to two times faster (see Table 3). Using node-local storage needs a redistribution of data from the storage on the single compute node used for decomposing the mesh to the 16 nodes needed for the solver. This data copy took 32 seconds, so even if not overlapped with other running tasks this approach would provide improved performance compared to directly using Lustre. The improvement is greater when run for a full simulation, which would require many thousands of timesteps, meaning the initial cost of copying the data would be negligible. There are several places where we have performance advantages: reduction of interferences, and better reading and writing of the data. On the other hand, the performance improvement depends on Lustre (11) (or PFS) performance, which is quite good in the prototype.

5 Filesystems Developed and Adapted for the Prototype

In NEXTGenIO we adopted GekkoFS, a distributed ad-hoc filesystem. GekkoFS allows the use of the NVDIMMs to create a Distributed File System (DFS), which is faster than going to the general (and shared) PFS. In our case, metadata operations are faster than on the PFS as we relax some POSIX compliance rules which are typically not needed.

Another development is a modification of the dataClay object store (see Section 5.2), designed with NVRAM in mind as it allows objects and methods to be stored together.

5.1 Block Level Storage

5.1.1 Distributed FS – GekkoFS

GekkoFS is a collaboration between BSC and University of Mainz. For use within NEXTGenIO we added POSIX capabilities to GekkoFS. GekkoFS allows the creation of a collaborative burst buffer using all the NVRAM of the compute nodes allocated by Slurm and tries to reduce the cost of distribution and finding data blocks using a hash function. Metadata is handled especially fast: GekkoFS is able to create 750,000 files per second (using 10 nodes).

The idea of this ad-hoc filesystem is to configure it with the minimum of characteristics required by the application. For example, we can remove locking if the application does not need it, or reduce security as the files will only be visible by the user.

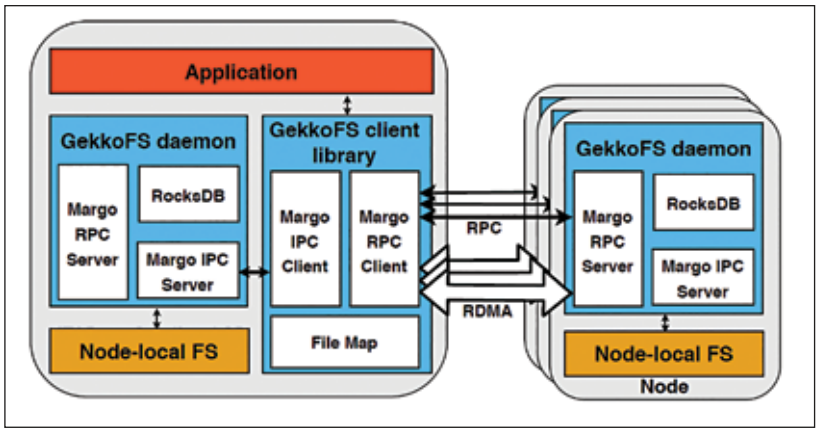

Figure 13: Overview of the GekkoFS system architecture. Applications use the traditional POSIX API to access a virtual collaborative burst buffer. The filesystem internally distributes I/O requests across available nodes.

The GekkoFS system architecture is shown in Figure 13. It consists of two main components: a client library and a server process. An application that uses GekkoFS must first preload the client interposition library through the LD PRELOAD environment variable which intercepts file system operations.

To limit the number of functions that have to be intercepted, GekkoFS uses the system call intercepting library (syscall intercept[4]) which provides a low-level interface for hooking Linux system calls, while still using the LD_PRELOAD method. As a result, the GekkoFS client only needs to intercept system calls, such as sys mknod or sys write, while leaving the functionality of, for instance, glibc untouched. Once the client has intercepted an I/O call, it forwards the I/O call to a server (GekkoFS daemon), if required. The GekkoFS daemon, which runs on each file system node, receives forwarded file system operations from clients and processes them, sending a response when finished. The daemons operate independently and do not communicate with other server processes on remote nodes, and are therefore effectively unaware of each other.

The latest version 0.6[5] is able to decouple servers and clients, so we can use more or fewer servers than clients. In addition, servers do not need to stay on the same node as clients.

[4] https://github.com/pmem/syscall_intercept

[5] https://github.com/NGIOproject/GekkoFS

5.1.1.1 Data Distribution

GekkoFS distributes data using a hash function among all the participating nodes (the default distribution is illustrated in Figure 14). Although this works for all applications, there are some applications that may improve performance by creating some special allocations. It is planned to offer such support via rules, for example, directory “node1” should have all the data local in node 1. This support will have an overhead, but we expect that the performance improvement will outweigh this.

Considering the example application OpenFOAM: it uses a distributed file system, but it has a local distribution: each processor writes in its own directory. So, in this scenario a data distribution that is tailored to the application will have significant benefits, such as a reduced communication between nodes and the possibility to use shared memory as local transport.

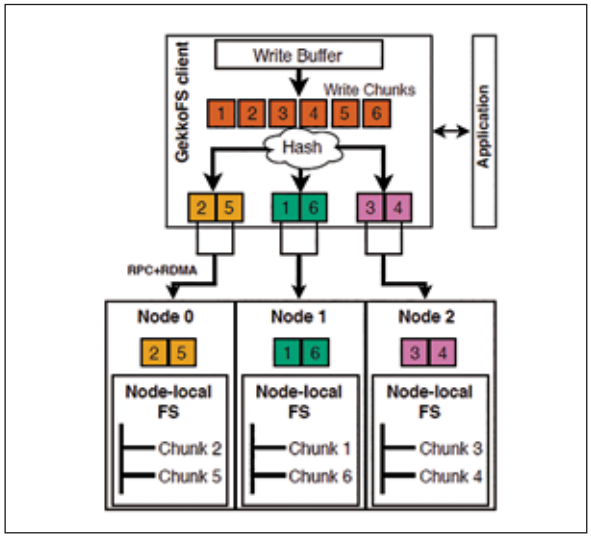

Figure 14: GekkoFS default data distribution

5.1.1.2 Slurm submission of a GekkoFS 0.6 Job

We enhanced Slurm in order to create filesystems on demand. GekkoFS is one of the supported filesystems. Users can run the following command:

```
# sbatch --nodes=2 --nvram-options=1LM:1000 --filesystem-type=gekkofs
--filesystem-device=/mnt/pmem_fsdax0
--filesystem- mountpoint=/home/nx01/nx01/ipanourg/gekkofstest runme.sh
```

This command will create all the pre-requisite files and environment variables and start the GekkoFS daemons. The daemons will be bound to a NUMA region; and users have the ability to only start one GekkoFS daemon per node. Once the GekkoFS daemons have started, Slurm will allow the job to run. After the job terminates, Slurm will shutdown the GekkoFS daemons. If any part of the GekkoFS configuration fails; the node will be put in the DOWN state in order to allow an administrator to check the issue.

5.2 Object Level Storage

5.2.1 What is dataClay and how can applications benefit from it?

dataClay[6] is a next generation object store where applications can store objects as they see the objects in the application's address space. The idea is that applications implemented using an object-oriented programming language (currently Java or Python) create complex data structures and when they want to make these data structures persistent, they just call a method `"obj.MakePersistent"`. From this point, any modification to the object 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 the application's 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's applications do. Current applications have no idea whether a piece of data is in L1 cache or elsewhere in memory. Secondly, data will be easily shareable between applications given that only the name of data needs to be passed from one application to another, much like a file name.

5.2.2 PyCOMPSs Integration

PyCOMPSs is a task-based programming model that enables the parallel execution of sequential Python applications with very small impact on the code. It relies on a set of decorators that allow the user to identify the functions/methods to be considered as tasks, a small API for synchronization, and a powerful runtime able to identify the data dependencies among tasks, extract their parallelism and manage their execution on distributed infrastructures. dataClay can annotate the Python code to mark the sections in the user's code that will be executed inside dataClay. Once PyCOMPSs executes a task, the dclayMethod will be executed in the object store.

The integration of PyCOMPSs and dataClay relies on an interface (storage API), that allows the PyCOMPSs runtime and users to communicate with dataClay, in order to perform the necessary storage actions. Since dataClay objects (Self-Contained Objects) can be used in any Python application, they can also be present in a PyCOMPSs application, and used as task parameters. SCOs can be persisted explicitly, which means that the object is stored within dataClay, so that its data and methods are no longer in main memory and the object is substituted with a reference. This reference is used by PyCOMPSs, avoiding management costs for the object, such as object serialisation, transfer, and deserialisation.

As the PyCOMPSs runtime schedules and runs the different tasks, it relies on well-known data versioning and replication techniques to avoid collisions and increase the parallelism. These actions are supported by dataClay through its API.

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

6 Summary

DCPMMs will be a key technology to advance memory and storage design, from both the hardware and software perspective. In this White Paper we explored NVRAM provided by Intel DCPMMs from the software perspective.

The large change from traditional storage devices (including SSDs) with latencies now near DRAM level, provides a big playground for applications. We need to provide applications with different modes (filesystem and memory) and evaluate 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) with a smaller number of options.

We have implemented and adapted storage systems to provide a set of tools that can be used by any application. The first of these systems is an ad hoc filesystem, able to provide local and distributed access to the NVRAMs. The second system is the object storage provided by **dataClay**, which is tuned specially to run with NVMs and is able to explore byte style addressing.

With regards to scheduling, we provide **Slurm** modifications and a new tool **NORNS** to provide asynchronous movements of data between different backends. This work will continue beyond the end of the NEXTGenIO project: **GekkoFS** will be used in different projects and we are working on a modification of this filesystem to support Java (as well as the COMPSs runtime).

7 References

1. *GekkoFS - A Temporary Distributed File System for HPC Applications*. **Vef, Marc-André, et al.** 2018. IEEE International Conference on Cluster Computing (CLUSTER).

2. *DAOS for extreme-scale systems in scientific applications*. **Breitenfeld, M. S., et al.** 2017. arXiv preprint arXiv:1712.00423.

3. *Slurm: Simple linux utility for resource management.* **Yoo, Andy B, Jette, Morris A and Grondona, Mark.** s.l. : Springer, 2003. Workshop on Job Scheduling Strategies for Parallel Processing. pp. 44-60.

4. **The Open Group; IEEE Computer Society; ISO/IEC JTC 1.** IEEE Std 1003.1™-2017 (Revision of IEEE Std 1003.1-2008). [Online] https://pubs.opengroup.org/onlinepubs/9699919799/.

5. **Free Software Foundation.** Linux Programmer's manual: malloc, free, calloc, realloc - allocate and free dynamic memory. [Online] http://man7.org/linux/man-pages/man3/malloc.3.html.

6. *A scalable concurrent malloc (3) implementation for FreeBSD*. **Evans, Jason.** Ottawa : s.n., 2006.

7. **Intel.** MEMKIND. [Online] https://github.com/memkind/memkind.

8. —. Persistent Memory Development Kit - The libvmmalloc library. [Online] https://pmem.io/pmdk/libvmmalloc/.

9. *OpenFOAM: A C++ library for complex physics simulations.* **Jasak, Hrvoje, Jemcov, Aleksandar and Tukovic, Zeljko.** Dubrovnik : IUC, 2007.

10. **Andrejašič, Matej and Veble, Gregor.** Cloud-based Simulation of Aerodynamics of Light Aircraft. [Online] https://hpc-forge.cineca.it/files/CoursesDev/public/2015/Workshop_HPC_Methods_for_Engineering/cloud_based_aircraft.pdf.

11. *The Lustre storage architecture.* **Braam, Peter.** 2002.