Dynamic Provisioning of Storage Resources: A Case Study with Burst Buffers

François Tessier, Maxime Martinasso, Matteo Chesi, Mark Klein, Miguel Gila
Swiss National Supercomputing Centre, ETH Zurich, Lugano, Switzerland
{firstname}.{lastname}@cscs.ch

***Abstract*—Complex applications and workflows needs are often exclusively expressed in terms of computational resources on HPC systems. In many cases, other resources like storage or network are not allocatable and are shared across the entire HPC system. By looking at the storage resources in particular, any workflow or application should be able to select both its preferred data manager and its required storage capability or capacity. To achieve such a goal, new mechanisms should be introduced. In this work, we present such a tool that dynamically provision a data management system on top of storage devices. We propose a proof-of-concept that is able to deploy, on-demand, a parallel file-system across intermediate storage nodes on a Cray XC50 system. We show how this mechanism can be easily extended to support more data managers and any type of intermediate storage. Finally, we evaluate the performance of the provisioned storage system with a set of benchmarks.**

***Keywords*-Dynamic provisioning; Software-defined; Storage; Data manager;**

I. Introduction

Large and complex scientific workflows such as generation of weather forecast data [1] or identification of new materials [2] define a set of tasks and dependencies among themselves. These workflows are expressed in terms of graphs of compute tasks, and, when running on a large and shared HPC system, data management of these workflows relies solely on accessing a global shared file system. Workflow data capability is therefore constrained by the provided parallel file system both for data format and performance variability. From a scientist point of view, expressing data-oriented tasks inside workflows enables a greater flexibility and a more complete definition of the workflow itself. However, on the HPC center side, it is not feasible to support for each system a large variety of data managers such as parallel file systems or databases [3]. Even more due to the economy of scale of resources, HPC centers tend to provide few and large data resources (but exclusive compute resources).

The past years have seen the amount of data generated by scientific workflows and large-scale HPC simulations dramatically increase. Despite attempts by vendors to temper this burden by deploying new tiers of memory and storage, it is clear that the performance gap between computing power and I/O operations continues to grow. Burst buffers for instance, such as Cray DataWarp [4], or hybrid storage tiers, such as NVMe disks, have been designed to mitigate the I/O slowdown by providing an intermediate tier of fast storage between the compute nodes and the parallel file system.

To expose new data managers or new data accesses on HPC systems, the trend is to multiply layers [5] inside the I/O software stack (specialized stack for shared resources, databases over file system, etc...). Such layers limit the capability to fully exploit the I/O hardware performance. Application developers are dependent on the software stack deployed on the system. For example, the Cray Data Virtualization Service (DVS) [6] which does a projection of the DataWarp storage onto the compute nodes, is necessary to use DataWarp nodes. On-node disk is another example of storage layer whose use is limited to the deployed file system.

In this work, we propose to dynamically provision HPC storage resources for workflows and applications. As for computing resources, data resources are managed as a batch scheduling resource and are requested during the job submission of the application or workflow task. The data manager is selected by the user inside the job scripts. The flexibility offered by such a dynamic provisioning mechanism is the key aspect of this work.

We first describe the design of such a component and the challenges incurred by its implementation and its integration in the workflow execution stack. We give a general overview of the requirements and limitations this type of service raises. Then, as a concrete example, we introduce a proof of concept that dynamically deploys BeeGFS [7] on a set of Cray DataWarp nodes. We first repurpose DataWarp nodes with compute node images which allows us on one hand to configure DataWarp as an allocatable resource in SLURM [8] and on the other hand to configure the raw storage devices for any data managers. In a second step, we enable the capability to deploy on-demand a well-sized BeeGFS on those DataWarp storage devices. Moreover, we show the portability of our method on non-Cray solutions hosting NVMe disks. We validate the reliability of this architecture by subjecting it to a high I/O load through benchmarks typical of workloads running on HPC systems.

The key contributions of our work are the following:

- A simple and portable mechanism to deploy on-demand data managers on top of intermediate storage resources;
- A proof of concept showing a new usage of Cray DataWarp as a storage layer for on-demand deployment of a BeeGFS file system;

- A performance evaluation of the provisioned file system.

II. Context and Motivation

Traditional HPC centers focus on providing one highly performing flagship machine for a precise set of scientists of different domains. Oppositely, Cloud providers intend to give to anyone access to commodity computation and storage capability. Because of their extremely general user base, they have developed infrastructure-as-a-service (IaaS) technology to let users configure and deploy the system they require. One key element of the IaaS technology is the dynamic provisioning of resources: compute, network and storage.

It is common for HPC system to provide dynamic access to compute nodes through a batch scheduler, however, little has been done for dynamically provisioning storage and network resources. Such resources are traditionally shared among all users. To maximize performance many HPC techniques have been developed to minimize contention and congestion on these two resources. By taking an IaaS approach, in this work, we introduce the idea of dynamically provisioning storage resources on contemporary HPC hardware.

There are many advantages for dynamically provisioning storage resources in an HPC context. For workflows and applications it allows to define precisely data managers type and configuration that fit their needs. In-transit analysis workflows, for instance, could take advantage of this. It also allows to select the storage hardware to use. Models can be investigated to automatically pick the most suitable storage layer to exploit based on a trade-off between data locality and performance (such as capacity or bandwidth). Another key aspect of this approach concerns the isolation brought by limiting access to the data managers to the application or workflow. Storage resources are not necessarily shared anymore among users. For instance, a metadata sensitive application could deploy a fast metadata file system on close-to-compute flash storage [9] to maximize performance and to avoid metadata contention initiated by other applications running on the system.

III. Architecture

Our dynamic resource provisioning mechanism consists of deploying on demand, a data management system on raw intermediate storage. To do so, we identified a list of requirements for accessing the underlying storage and we designed our provisioning mechanism such as resources can be seamlessly requested through the job scheduler. For the rest of this paper, we will use the term *storage node* to refer to nodes with local storage. We also make here the assumption that nodes (storage and compute) are part of the same network and can be mutually reachable.

In this section, we will first present the main architecture of our dynamic resource provisioning mechanism. Then, we will list the prerequisites needed to use the intermediate storage layer and discuss some technical limitations that need to be addressed for using this tool on a production system.

A. Dynamic provisioning of storage resources

Figure 1 depicts how our dynamic resource provisioning mechanism works on a HPC system. Through the job scheduler, a user can request two allocations: the compute nodes needed to run the application(s) and a set of storage nodes to deploy a data manager. The set of storage nodes and the set of compute nodes can be disjoints (burst buffers) or the former can be a subset of the latter (compute node with local storage). It is important to note that our approach is independent of the job scheduler as long as it is possible to request an allocation of storage nodes. Once allocations have been granted, data management services are deployed on the storage nodes and clients, if any, are set up on the compute nodes. To insure a better portability of the tool across job schedulers, deploying a data manager is an extra command lines in a batch script. One could develop a job scheduler specific plugin but it will require more development and maintenance efforts.

On the allocated storage nodes, containers packaging the data management system are launched. A script is executed along with the container to configure and start the services. A containerized approach allows to provide data managers independently of the software stack installed on the system. Those services will remain accessible from outside of the container as the processes are visible in the PID namespace of the host. In addition, using containers offers a certain portability across different platforms, and opens up the possibilities to utilize the multitude of data management systems available on repositories such as DockerHub [1].

On the compute node side, it is necessary to configure access to the previously deployed data manager. Depending on the data management system, this step can be done by simply giving the master node's IP address or by a more complex manner with a daemon, a kernel module or a container if necessary. We will discuss these aspects in III-B.

As shown in Figure 1, this architecture offers two options to the running application or workflow to perform I/O. While the shared parallel file system is still accessible, a temporary data manager can now be accessed (yellow box "Data manager").

When the computing and storage resources are released by the user or the job scheduler, services on storage nodes are terminated and data on disks is deleted. A stage-in/stage-out mechanism is worth considering to back up data. Thereby, the cost of moving data has to be taken into account. On compute nodes, clients are also terminated.

[1] https://hub.docker.com/

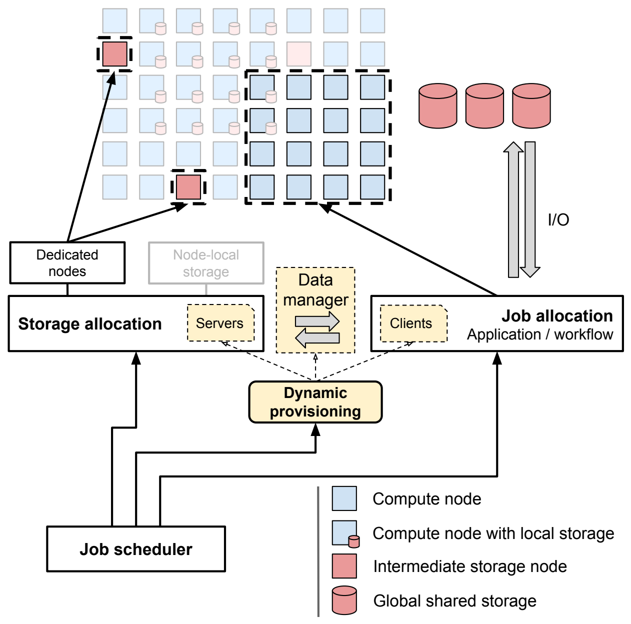

Figure 1. Dynamic provisioning of storage resources

B. *Requirements and limitations*

Our proposed approach for dynamically provision resources has a set of requirements and suffers some limitations. In particular, we identified four important items described hereinafter:

- For some types of intermediate storage technology, like burst buffers, system administration may be required to reconfigure the hardware and allow the deployment of an on-demand data manager. On a typical HPC system with node-local storage, disks are usually directly writable although the underlying file system cannot be changed. In case of burst buffers, however, intermediate storage is distributed across dedicated nodes and generally accessed through a specific I/O layer. Most of the time, users cannot access the nodes but, instead, can only handle data by means of a mount point. Granting access to those nodes is necessary to deploy a new data management system and by-pass the dedicated I/O interface.
- As briefly mentioned in III-A, when a storage is dynamically provisioned by a job, it does not contain any data. Therefore, a stage in and stage out of data might be required for the scientific application to run or to retrieve its results. Nevertheless, many high-end HPC systems offer a high-performance scratch file system for which such stage in and out steps are necessary. The only difference is that the movements of data in the latter case are not accounted in the job runtime.
- Another limitation that needs to be acknowledged relates to the level of privileges in the operating system that the data manager requires to be executed. Many file systems, for instance, need to run inside the kernel space. Sometimes, a kernel module is necessary on the client side to interact with the data management system. Fine-grained privileges to users is one workaround. Another solution could be to use *prolog* and *epilog* scripts executed by the job scheduler to load those modules. That way, escalation of privileges is avoided at a user level. The node image, however, sill need to have the kernel module pre-installed.
- Last, the notion of resources for storage can be either focusing on capacity (quantity of bytes) or capability (speed of read/write operations). In the later case, using more disks increases the overall I/O operation bandwidth. It means that if storage capability is targeted by the user, he or she should ask for more storage nodes possibly wasting disk capacity. In any case, many compute jobs also under-utilize their compute resources by using only a portion of the node memory or efficiently utilizing either an accelerator or the host processor.

IV. Proof of Concept

In this section, we will give implementation details of the proof of concept of our design introduced in the previous section. We will show how we set up our test-bed and how we were able to deploy on-demand a BeeGFS parallel file system on top of SSD-based burst buffers located inside Cray DataWarp nodes.

A. *Prerequisites for accessing intermediate storage*

In order to grant a high level of interaction with DataWarp nodes and their resources, it is essential to bypass the DVS (for Data Virtualization Service) layer. Thus, we repurposed the DataWarp nodes from hidden service nodes to standard compute nodes with a system customization. We then configured their local NVMe storage. The repurposing consisted in two reconfiguration changes in the Cray XC system configuration database. We first modified the node type from `service` to `compute` through the `xtprocadmin` command then we mapped a compute node image to boot with via the `cnode update` CLI tool. The additional configuration changes needed were made through Ansible, a popular system administration tool for distributed deployment, and consisted in formatting flash devices with a XFS file system and mount them on the node with all permissions granted to any user.

While local storage of compute nodes is accessible with a standard allocation, the Cray DataWarp technology provides a SLURM Burst Buffer plugin interface to allocate DataWarp storage resource. This interface is well suited for a standard Burst Buffer implementation to do check-pointing, but it limits the user interaction with storage resources to those functionalities already implemented in the interface. A strong coding effort is required to change or enhance this plugin. In order to overcome these limitations of using such SLURM Burst buffer plugin, the re-purposed DataWarp nodes are made available to end users through a SLURM constraint. In the same way users can use `gpu` constraint to request nodes equipped with GPU or `mc` constraint for a multicore compute node on CSCS systems, the storage nodes are provided requesting `storage` constraint.

B. Implementation details

The proof of concept of our dynamic provisioning method has been implemented with Python scripts packaged in a Docker [10] container along with the software stack necessary for the data manager. A Bash script is in charge of deploying the container on all the granted storage nodes and a Python script is executed to start the data management services. The containers are deployed using Shifter [11], [12], a container engine for HPC systems. The list of available disks and their mount points on each node are described in a configuration file which is accessible from the container. On the compute nodes side, scripts are executed on each node to set up the client, if any. This deployment phase, as it consists of running two bash scripts (on the server side and on the client side) can be carried out from a batch script or within an interactive session on nodes.

To validate our dynamic provisioning method, we used BeeGFS as a data manager. BeeGFS is a POSIX-compliant parallel file system with a client-server architecture. It features five main components:

- A management server in charge of orchestrating the other daemons;
- At least one metadata server for metadata;
- At least one storage server for raw data;
- A monitoring service accessible from a desktop Java application;
- A BeeGFS kernel module for the client.

The BeeGFS container deployed on storage nodes contains a fresh install of all the packages required to start the parallel file system. It also includes a Python script set as the entry point of the container (it is executed at container launch time). It is in charge of creating all the configuration files for each server-side component of BeeGFS: management, metadata, storage and monitoring. Particularly, it sets up the network parameters (IP of the management server, communication ports), the absolute path of the mount point of the disk that will be used by the service (*/mnt/nvme0n1* for metadata for instance) and a few other daemon-specific settings like, for example, the capability to use file system extended attributes for metadata. The script also starts all the daemons within the container in user-space. The service distribution across the disks is fixed in the current implementation. The first disk of the first node hosts the management daemon, as well as the monitoring service. Then, on each storage node, the first disk receives a metadata server (so, co-located with other services on the first node) while the two other SSDs are exploited for storage purpose.

On the compute nodes, a script initializes the BeeGFS client configuration and creates a local mount point of the running BeeGFS instance. This last step incurs the need of a kernel module to mount the file system and consequently it implies that the kernel module is present on the node image. In addition, the script requires privileges to locally mount BeeGFS. On the experimentation platforms we used to evaluate our dynamic resource provisioning mechanism, privileges have been escalated to let us configure this setup. As explained in III-B, to overcome these restraints on production systems, we plan to investigate a solution where the job scheduler setups the environment during prolog execution, loads the kernel module and then builds and mounts the file system. In the same way, the epilog script carried out at the end of the job can unmount and delete the file system and finally unload the kernel module.

Whether it is our tool with appropriate user privileges or a SLURM prolog script, mounting a running BeeGFS file system is a simple command line such as:

```
$ mount -t beegfs beegfs_nodev \
        <mount_point> \
        -ocfgFile=beegfs-client.conf,_netdev,,
```

From an application point of view, BeeGFS is accessed by writing or reading data to/from the locally mounted point.

V. Evaluation

We present in this section a performance evaluation of the proposed solution on two systems. First, we targeted a Cray system with DataWarp nodes. We ran on this machine multiple tests with IOR [13] to cover typical I/O workloads. We also carried out experiments with HACC-IO [14], the I/O kernel of a large-scale cosmological application. Secondly, in order to show the portability of our approach, we ran experiments on a compute node equipped with local NVMe disks.

A. Cray XC50 with Cray DataWarp

We first deployed our dynamic provisioning mechanism on Dom, a Cray XC50 system. Dom is the test and development system of Piz Daint, a 27 PFlops XC50 supercomputer at CSCS. The testbed features 8 multicore nodes, each with two 18-core Intel Broadwell CPUs (Intel Xeon E5-2695 v4) and 64 GB of DRAM. Within the Cray Aries network connecting the compute nodes, we also find 4 DataWarp nodes each embedding three 5.9TB PCIe SSD (Samsung PM1725a) whose vendor's value for I/O bandwidth is 6.2GBps for sequential read and 2.9GBps for sequential write [2]. Our experiments using the `dd` tool

[2]https://www.samsung.com/semiconductor/ssd/enterprise-ssd/MZPLL6T4HMLA-00005/

with multiple concurrent streams showed empirical peak performance values for reading and writing of respectively 6.34GBps and 3.2GBps. The global storage system on Dom provides 170TB of usable space managed by a Lustre file system [15] and distributed across 2 OSTs (object storage target). As being a test and development machine not opened to the users, we can consider Lustre as a dedicated parallel file-system.

For all of our experiments on Dom, we will use a number of DataWarp nodes and disks that provides performance in the same order as the Lustre configuration. We used two disks for storage and one disk for metadata per DataWarp node. The disk dedicated to metadata on the first node of the storage allocation was also used for BeeGFS management and monitoring. For Lustre, we set the stripe count (number of OST used to stripe files across) to 2. For the benchmarks presented below, we used the 8 compute nodes with 36 processes per node (288 processes total) for all the runs. The stripe size for both file systems was set to 1MB.

1) Deployment time: The time to deploy a containerized file system on multiple nodes can have large variations. Many factors can delay the deployment such as reconnecting the management server and the metadata due to network connection failures. Our experiments, however, showed an average deployment time of 5.4 ($\pm$ 1.2) seconds over three runs for a deployment of BeeGFS on two DataWarp nodes.

2) IOR: The IOR [13] suite is a popular and highly tunable set of I/O benchmarks, especially used for the IO-500 ranking [16]. It is composed of *IOR* for evaluating I/O performance and *mdtest* for appraising metadata management. We evaluated these two metrics on the on-demand BeeGFS file system and on Lustre. For IOR, we focused our experiments on two ways of writing data out: a single shared file or one file per process. As our goal was to show how the file systems can mitigate the burden caused by multiple streams, we performed independent MPI-IO calls instead of collective operations. We also set flags to disable client-side cache effect. We ran all the experiments ten times.

Figure 2 compares the I/O bandwidth attained when the 288 processes write then read back a single shared file to/from the on-demand BeeGFS deployed over two DataWarp nodes and the Lustre file system. We first observe than the write bandwidth is comparable from 32MB per process written into the shared file and beyond. Both file systems achieve around 6GBps. With smaller sizes however, Lustre outperforms BeeGFS but at the cost of a significant variability. When reading back data, BeeGFS on two DataWarp nodes performs approximately 2x better than Lustre and even more with 4MB per process. Nevertheless, when reading back 512MB or 1GB per process from our on-demand BeeGFS, the read bandwidth dramatically decreases. We explain this behavior by the fact that the BeeGFS caching mechanism size is limited to the 64GB of DRAM on each DataWarp node. With a balanced I/O load on the two nodes used here to deploy BeeGFS, the amount of data to manage per storage node is $\frac{1}{2} \times \#computenodes \times \#ppn \times S_p$, S_p being the size of the data written or read per process. For $S_p \geq$512MB, this value is greater or equal to 73.72GB and the cache cannot fit in the available memory. The benefit of a server-side caching is therefore highly reduced.

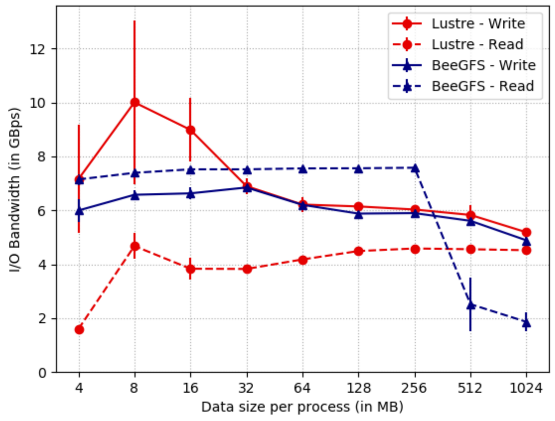

Figure 2. I/O bandwidth achieved on Dom with the IOR benchmark running on 8 compute nodes (36 ppn) according to the data size written per process into a **single shared file**. Comparison between on-demand BeeGFS deployed across 2 Cray DataWarp nodes and Lustre with 2 OSTs.

We depict on Figure 3 the same experiments with one file per process written and read back instead of a single shared file. This file distribution is known to provide better I/O performance. Except for a few data sizes, the dynamically provisioned BeeGFS reaches an higher I/O bandwidth than Lustre for both writing and reading. When reading back data, we observe a similar behavior as described previously: with large sizes (512MB and 1GB per process), the I/O bandwidth is low. An unexpected behavior appears on Lustre when writing 4MB files. This level of performance cannot be achieved in practice, which leads us to believe that a write cache effect that we have not been able to contain is at work for this particular data size.

Another remark is that the peak write bandwidth recorded with BeeGFS (with 64MB/process) is 70% higher than the peak bandwidth observed on a single shared file (11.96GBps against 7.01GBps). If we consider the cumulative write performance of the four storage disks of 12.8GBps total plus the metadata managed by two daemons on dedicated disks, we can conclude here that the file system is being used at the maximum of its capability.

The scalability of a dynamic resource provisioning system depends on multiple factors, both in terms of underlying hardware (number of nodes and disks) and storage software layers. Although our resources were limited we tried to

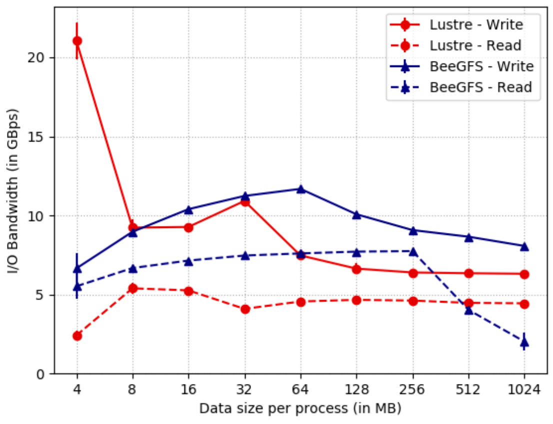

Figure 3. I/O bandwidth achieved on Dom with the IOR benchmark running on 8 compute nodes (36 ppn) according to the data size written per process into its own file (**one file per process**). Comparison between on-demand BeeGFS deployed across 2 Cray DataWarp nodes and Lustre with 2 OSTs.

evaluate the scalability of our solution. To do so, we ran the IOR benchmark from the 8 compute nodes while varying the size of the on-demand BeeGFS from 1 node to 4 DataWarp nodes. We kept a ratio of 1:2 of metadata:storage servers per node. Figure 4 presents those results for both types of file distribution. As expected, the scalability is satisfying for almost all the cases. Only the performance of the write bandwidth with a single shared file follows a logarithmic curve: the write bandwidth almost triples from 1 to 2 DataWarp nodes but increases by only 30% when doubling again the number of storage nodes.

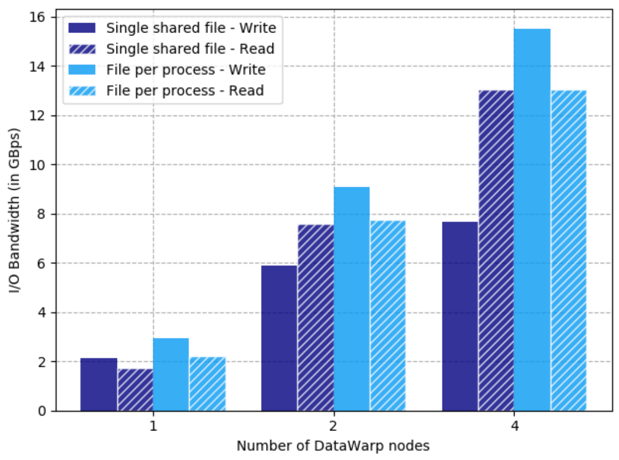

Figure 4. I/O bandwidth achieved on Dom with the IOR benchmark running on 8 compute nodes (36 ppn) while varying the size of the dynamically deployed data manager.

3) mdtest: The *mdtest* benchmark has been designed to evaluate the metadata performance of a parallel file-system. Directories and files are created, stated and removed multiple times and the number of operations per second is measured for each operation. Table I shows the results of this benchmark on the BeeGFS over two DataWarp nodes and on the Lustre file system. Except when stating a directory, Lustre metadata management outperforms the dynamically provisioned BeeGFS. File creation for instance is 3.5x faster on Lustre. The value obtained with BeeGFS for directory stat looks very high. A client-side cache probably explains this result.

		BeeGFS	**Lustre**
Target	**Operation**	**Ops**	
Directory	creation	8276.43	37222.57
	stat	5301788.76	182330.42
	removal	12967.02	38732.00
File	creation	6618.37	22916.15
	stat	144410.46	169140.32
	read	22541.08	45181.55
	removal	8431.71	35985.96
Tree	creation	2183.40	3310.42
	removal	125.23	1298.55

Table I
I/O OPERATIONS PER SECOND FOR VARIOUS OPERATIONS PERFORMED ON THE DYNAMICALLY PROVISIONED BEEGFS AND ON LUSTRE WITH THE MDTEST BENCHMARK ON DOM FROM 8 NODES (36 PPN).

4) HACC-IO: HACC-IO is the I/O kernel of HACC [14] (Hardware Accelerated Cosmology Code). This large-scale cosmological application, developed at Argonne National Laboratory, requires the massive compute power of super-computers to simulate the mass evolution of the universe with particle-mesh techniques. In terms of I/O, every process of a HACC simulation manages a number of particles. Each particle is defined by nine variables—XX, YY, ZZ, VX, VY, VZ, phi, pid, and $mask$—corresponding to the coordinates, the velocity vector, and relevant physics properties. The size of a particle is 38 bytes. A useful base value of 25,000 particles requires approximately 1 MB. The data layout of those particles in file follows an array of structure pattern as described in Figure 5. Data is written and read back from a single shared file by all the processes.

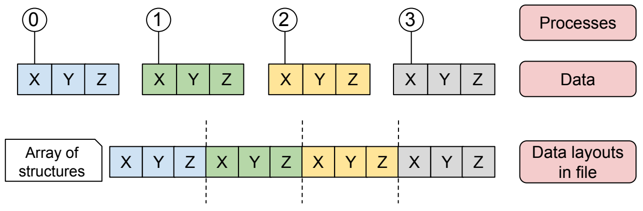

Figure 5. Array of structure data layout implemented in HACC-IO

Similarly to the previous experiments, we ran HACC-IO on 8 compute nodes (36 ppn) on Dom and compared the I/O performance of the two file systems, BeeGFS and Lustre, respectively using two DataWarp nodes and two OSTs. We see on Figure 6 that the on-demand file system offers the best read and write bandwidth up to a 42GB file size. The peak write bandwidth is 5.3GBps while data is read back up to 9.1GBps. The Lustre file system, however, performs poorly. 1GBps is barely attained during the write phase. The read bandwidth stays below 0.4GBps regardless the input file size. Previous work [17] evaluating HACC-IO on Lustre corroborates those results.

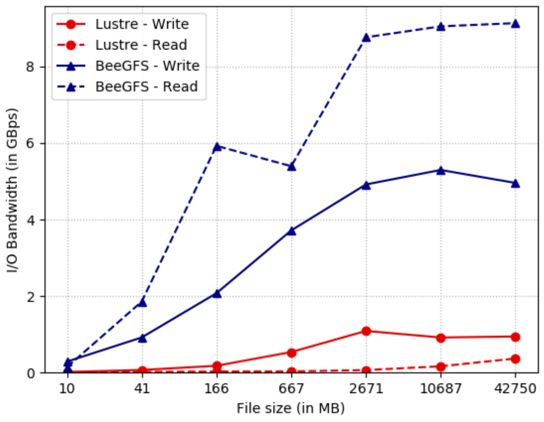

Figure 6. I/O bandwidth achieved on Dom with the HACC-IO running on 8 compute nodes (36 ppn) according to the number of particles (data size) per process into a **single shared file**. Comparison between on-demand BeeGFS deployed across 2 Cray DataWarp nodes and Lustre with 2 OSTs.

B. Non-Cray testbed

To assess the portability of the proposed approach, we conducted experiments on another system, called Ault. Ault is test-bed platform at CSCS that allows for prototyping experimental services and platforms. Various types of hardware are available for researchers to quickly provision as needed using Canonical's Metal as a Service (MaaS) product. This allows for safe privileged-access level experimentation by researchers without impacting production services. For our experiments, we used Ault11, a compute node with a 22-core Intel Xeon Gold 6152 CPU cadenced at 2.10GHz. The node also hosted 16 NVMe disks (Intel SSD DC P4500) whose vendor's value for sequential read and write is respectively 3.2GBps and 1.9GBps [3]. As for the disks on DataWarp nodes, this performance values do not reflect a real use-case with multiple concurrent streams.

We set up the following configuration for the on-demand file system:

- 1 disk for BeeGFS management and monitoring
- 2 disks for the metadata
- 5 disks for distributed storage

1) Deployment time: On Ault, the deployment time for the BeeGFS instance across 8 empty disks is approximately 4.6 seconds. If the tree structure of the file system has been written already, this initialization phase decreases to 1.2 seconds.

2) IOR: We ran on this platform the same IOR experiments as in V-A2 adapted to the number of available computing and storage resources. Figure 7 depicts the results of the IOR benchmark on the node-local on-demand BeeGFS with a single shared file and one file per process. Results are in correlation with the empirical performance we can expect for those disks with multiple concurrent streams. The peak read bandwidth attained 20.36GBps following a file-per-process distribution while, for the same file division, the peak write bandwidth reached 13.70GBps. Again, performing I/O into dedicated files per process substantially increases the I/O bandwidth.

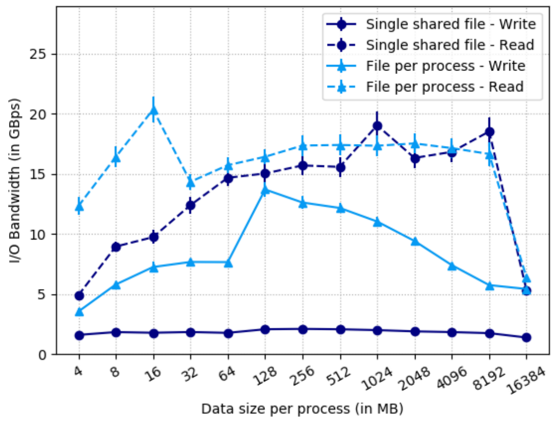

Figure 7. I/O bandwidth achieved on Ault with the IOR benchmark running on one compute node (22 ppn) according to the data size written per process. The on-demand BeeGFS is deployed on 8 NVMe disks.

3) mdtest: Table II presents the metadata performance of the on-demand BeeGFS distributed across 8 NVMe disks on Ault. Results are quite far from what a raw disk can theoretically absorb. However, it is in line with previous results on Dom for a similar configuration (2 nodes dedicated to metadata).

VI. RELATED WORKS

The Cloud community has introduced the concept of IaaS. Technology such as OpenStack [18] or Kubernetes [19] allows to create virtual clusters on hardware resources. Even if both tools are technically different, their goals is to define virtual clusters by grouping amount of compute, network and storage resources. For OpenStack the storage resources could include either a block storage, an object storage,

[3]https://ark.intel.com/content/www/us/en/ark/products/128484/intel-ssd-dc-p4500-series-8-0tb-ruler-pcie-3-1-x4-3d1-tlc.html

Target	Operation	Ops
Directory	creation	1796.31
	stat	667250.43
	removal	5516.92
File	creation	5234.87
	stat	98888.28
	read	22889.51
	removal	5929.99
Tree	creation	2754.81
	removal	980.84

Table II
I/O OPERATIONS PER SECOND FOR VARIOUS OPERATIONS PERFORMED ON THE DYNAMICALLY PROVISIONED BEEGFS WITH THE MDTEST BENCHMARK ON AULT FROM ONE NODE (22 PPN).

or ephemeral storage. Kubernetes increased the range of storage with a large list of possible options. However, none of them comprise integrated HPC storage file systems such as Lustre [20] or IBM Spectrum Scale [21] (formerly GPFS).

In parallel, Cloud providers are starting to propose HPC storage capability. Amazon Web Services (AWS) [22] is leading this effort. Recently, AWS offers the ability to deploy an on-demand Lustre file system [23]. AWS also offers the IBM Spectrum Scale [24] technology. However, such deployment still requires human intervention. In the AWS ecosystem, traditional HPC-oriented file systems are not solely limited to a one-time global installation but have the possibility to be dynamically provisioned.

Some HPC-oriented file systems are starting to provide on-demand provisioning features. BeeGFS [7], for instance, provides the BeeOND [25] option. By using a single script a full instance of the file system can be instantiated. Our implementation of the deployment of BeeGFS across storage nodes is quite similar to the bash script behind BeeOND. Nonetheless, although our version follows the same steps, it provides more flexiblity and generality tends to be extended to other data managers (database, object store, and so on). More recently, work has been done to develop an "on-demand" capability in Lustre [26]. Yet, the wish to provide storage dynamically is not new. Wickberg et al. [27], for example, already proposed a few years ago an ephemeral file system deployed on the fly and on a per-job basis. All of these solutions, however, lack flexibility, both in terms of hardware resources they support and the often unique data manager they can deploy.

If we look at what has been done on burst buffers like Cray DataWarp or DDN Infinite Memory Engine [28], it is interesting to note that the idea of temporary file system have already been developed [29] [30]. However, such work propose a new file system software with the feature to be deployed on-demand on a burst-buffer device. Our work provides a more generic solution in terms of data managers and target storage hardware.

VII. CONCLUSION

We presented in this paper the design and a proof of concept of a mechanism to dynamically provision a data manager on top of intermediate storage resources. Such an approach allows an application or a workflow to get, on-demand, an isolated data management system meeting its requirements. In a period of HPC/Cloud convergence, such a technique brings the flexibility of a Cloud environment onto performance of an HPC system.

As an example, we focused on deploying BeeGFS, a parallel file system, over burst buffer nodes using the Cray DataWarp technology. We evaluated the deployed file system and compared it to a global storage system on a small-scale platform. Our experiments showed good performance and scalability for our method. We also proved the portability of our container-based tool on another system equipped with a different storage technology.

This preliminary work will be largely extended in the future. We plan to strengthen the current implementation to simplify the support of new data managers. We will improve the configuration capability of our system. Another improvement will be to minimise the requirements of containerized data manager in such a way that they can be pulled from their official repository and be used without any change. Finally, cost models for automatic deployment based application requirements such as capacity or bandwidth will be another area to explore.

ACKNOWLEDGMENT

This research is supported by the European project Maestro[4]. The Maestro project has received funding from the European Union's Horizon 2020 research and innovation program through grant agreement 801101.

REFERENCES

[1] A. Bonanni, S. Smart, and T. Quintino, "Kronos: Benchmarking HPC systems with realistic workloads." [Online]. Available: http://www.nextgenio.eu/publications/kronos-benchmarking-hpc-systems-realistic-workloads

[2] G. Pizzi, A. Cepellotti, R. Sabatini, N. Marzari, and B. Kozinsky, "AiiDA: automated interactive infrastructure and database for computational science," *Computational Materials Science*, vol. 111, pp. 218 – 230, 2016.

[3] J. Lüttgau, M. Kuhn, K. Duwe, Y. Alforov, E. Betke, J. Kunkel, and T. Ludwig, "Survey of storage systems for high-performance computing," *Supercomputing Frontiers and Innovations*, vol. 5, no. 1, 2018. [Online]. Available: http://www.superfri.org/superfri/article/view/162

[4] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright, "Architecture and design of Cray DataWarp," in *Proceedings of 2016 Cray User Group (CUG) Meeting*, 2016.

[4]https://www.maestro-data.eu/

[5] R. Ross, L. Ward, P. Carns, G. Grider, S. Klasky, Q. Koziol, G. K. Lockwood, K. Mohror, B. Settlemyer, and M. Wolf, "Storage systems and i/o: Organizing, storing, and accessing data for scientific discovery," *Report for the DOE ASCR Workshop on Storage Systems and I/O*, 9 2018.

[6] S. Sugiyama and D. Wallace, "Cray dvs: Data virtualization service," in *Cray User Group Annual Technical Conference*, 2008. [Online]. Available: https://cug.org/5-publications/proceedings_attendee_lists/2008CD/S08_Proceedings/pages/Authors/16-19Thursday/Wallace-Thursday16B/Sugiyama-Wallace-Thursday16B-paper.pdf

[7] "BeeGFS," https://www.beegfs.io/content/, accessed: 2019-04-12.

[8] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60.

[9] J. Kim, H. Jo, H. Shim, J.-S. Kim, and S. Maeng, "Efficient metadata management for flash file systems," in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, 2008, pp. 535–540.

[10] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[11] L. Benedicic, F. A. Cruz, A. Madonna, and K. Mariotti, "Portable, high-performance containers for HPC," *CoRR*, vol. abs/1704.03383, 2017. [Online]. Available: http://arxiv.org/abs/1704.03383

[12] L. Gerhardt, W. Bhimji, S. Canon, M. Fasel, D. Jacobsen, M. Mustafa, J. Porter, and V. Tsulaia, "Shifter: Containers for hpc," *Journal of Physics: Conference Series*, vol. 898, p. 082021, 10 2017.

[13] "IOR: Parallel filesystem I/O benchmark," https://github.com/hpc/ior, accessed: 2019-04-12.

[14] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka, V. Vishwanath, Z. Lukić, S. Sehrish, and W. keng Liao, "HACC: Simulating sky surveys on state-of-the-art supercomputing architectures," *New Astronomy*, vol. 42, pp. 49 – 65, 2016.

[15] "Lustre filesystem website," http://lustre.org/.

[16] J. Kunkel, G. Markomanolis, J. Bent, and J. Lofstead, "VI4IO/io-500-dev: Zenodo citation release," Sep. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1422814

[17] F. Tessier, P. Gressier, and V. Vishwanath, "Optimizing data aggregation by leveraging the deep memory hierarchy on large-scale systems," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. New York, NY, USA: ACM, 2018, pp. 229–239. [Online]. Available: http://doi.acm.org/10.1145/3205289.3205316

[18] A. Shrivastwa, S. Sarat, K. Jackson, C. Bunch, E. Sigler, and T. Campbell, *OpenStack: Building a Cloud Environment*. Packt Publishing, 2016.

[19] K. Hightower, B. Burns, and J. Beda, *Kubernetes: Up and Running Dive into the Future of Infrastructure*, 1st ed. O'Reilly Media, Inc., 2017.

[20] P. Schwan, "Lustre: Building a file system for 1,000-node clusters," in *Proceedings of the Linux Symposium*, 2003, p. 9.

[21] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters." in *FAST*, vol. 2, no. 19, 2002.

[22] G. Sammons, *Introduction to AWS (Amazon Web Services) Beginner's Guide Book: Learning the Basics of AWS in an Easy and Fast Way*. USA: CreateSpace Independent Publishing Platform, 2016.

[23] "Amazon FSx for Lustre," https://aws.amazon.com/fsx/lustre.

[24] "IBM Spectrum Scale on AWS," https://aws.amazon.com/quickstart/architecture/ibm-spectrum-scale.

[25] F. Herold, S. Breuner, and J. Heichler, "An introduction to BeeGFS," 2014.

[26] S. Ihara and R. Deshmukh, "Lustre On Demand: Evolution of Data Tiering on Storage System," https://www.eofs.eu/_media/events/lad18/08_rahul_deshmukh_lad18_lustre_on_demand_si_rd_final2.pdf.

[27] T. Wickberg and C. Carothers, "The ramdisk storage accelerator: a method of accelerating i/o performance on hpc systems using ramdisks," *ROSS 2012*, 06 2012.

[28] "DDN - Infinite Memory Engine," https://www.ddn.com/products/ime-flash-native-data-cache/.

[29] M. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "GekkoFS - a temporary distributed file system for hpc applications," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 319–324.

[30] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An ephemeral burst-buffer file system for scientific applications," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2016, pp. 807–818.