MemEFS: A Network-aware Elastic In-Memory Runtime Distributed File System

Alexandru Uta[a,*], Ove Danner[a], Cas van der Weegen[a], Ana-Maria Oprescu[a,b,*], Andreea Sandu[a], Stefania Costache[a], Thilo Kielmann[a,*]

[a] *Dept. of Computer Science, Vrije Universiteit Amsterdam, The Netherlands*
[b] *Informatics Institute, Universiteit van Amsterdam, The Netherlands*

Abstract

Scientific domains such as astronomy or bioinformatics produce increasingly large amounts of data that need to be analyzed. Such analyses are modeled as scientific workflows - applications composed of many individual tasks that exhibit data dependencies. Typically, these applications suffer from significant variability in the interplay between achieved parallelism and data footprint. To efficiently tackle the data deluge, cost effective solutions need to be deployed by extending private computing infrastructures with public cloud resources. To achieve this, two key features for such systems need to be addressed: *elasticity* and *network adaptability*. The former improves compute resource utilization efficiency, while the latter improves network utilization efficiency, since public clouds suffer from significant bandwidth variability. This paper extends our previous work on MemEFS, an in-memory elastic distributed file system by adding *network adaptability*. Our results show that MemEFS' elasticity increases the resource utilization efficiency by up to 65%. Regarding the network adaptation policy, MemEFS achieves up to 50% speedup compared to its network-agnostic counterpart.

Keywords: in-memory file system, distributed hashing, elasticity, scalable computing, network variability, network adaptation, high-performance I/O, large-scale scientific computing, big data and HPC systems, big data for e-Science, large-scale systems for computational sciences

1. Introduction

An important direction of eScience research focuses on *efficiently* running *scientific workflows*. These computations are typically composed of many data-intensive tasks, spanning multiple domains, such as astronomy [1] and bioinformatics [2]. Such workflows often express inter-task dependencies by means of files (i.e. the output of one task is the input of another), containing (intermediary) data, generated at runtime. In contrast to traditional message passing mechanisms [3], this communication scheme requires a *shared*, often *distributed*, file system. However, *data-intensive* scientific workflows generate large data amounts, which cannot be efficiently handled by traditional, disk-based distributed file systems, thus leading to limited application performance and scalability.

To alleviate the storage bottleneck, the state-of-the-art [4, 5] suggests using in-memory runtime distributed file systems. Such systems either use a locality-based approach [4] or are locality agnostic [6, 5]. These solutions expose the compute nodes' memories as a fast, unified, distributed cache, that optimizes accesses to runtime-generated data. However, as in-memory runtime distributed file systems are typically *statically deployed* onto a fixed number of compute nodes, their *applicability* and *efficiency* are *limited*. For instance, the user would be faced with the difficult task of estimating the storage demands of the application. An under-estimation would lead to poor performance (as the system would start swapping), or even to crashing, while an over-estimation would lead to *poor resource utilization*.

Furthermore, *scientific workflows* exhibit significant variability in the data footprint and in the achieved parallelism [7]. The former is determined by two aspects:

*Corresponding author
Email addresses: a.uta@vu.nl (Alexandru Uta), o.a.danner@student.vu.nl (Ove Danner), c.vander.weegen@student.vu.nl (Cas van der Weegen), a.m.oprescu@vu.nl (Ana-Maria Oprescu), a.sandu@vu.nl (Andreea Sandu), s.v.costache@vu.nl (Stefania Costache), thilo.kielmann@vu.nl (Thilo Kielmann)

(i) data-intensive parallel stages generate large amounts of data; (ii) intermediary runtime generated data can be discarded when it is no longer needed. The latter occurs as a result of the mix of large parallel stages with sequential synchronization points, e.g., data aggregation, data partitioning stages. Clearly, a static deployment scheme needs to over-provision resources to accommodate for peak storage demands of the application. In shared clusters, such a behavior translates to longer queuing times for users - resources are needlessly reserved, and/or inefficient energy consumption - assuming idle machines could be powered off.

Ideally, an *elastic* in-memory runtime distributed file system would acquire and release resources according to application demands, thus offering more *flexibility*. A direct benefit of an *elastic in-memory storage* is that the user can rely on the system to determine the (near-) optimal number of nodes that an application needs for storing its runtime data. Also, nodes could be added or removed on-demand, during runtime.

Limiting the elastic approach to private computing infrastructure capacity, however, is ill-suited for the rapid increase in the data volumes produced by typical scientific applications [8]. Ideally, the private computing infrastructure would be augmented by means of on-demand, public cloud resources.

Here, a new type of problem appears: in contrast to private computing infrastructure, many studies [9, 10] point out that in public clouds the network performance is impacted by large degrees of variability - due to virtualization, colocation and congestion overheads [11, 12].

This paper introduces a data distribution policy proportional to network capabilities that aims to better utilize compute and network resources. The key insight of the proportional policy is that the interplay between compute-to-storage ratio and available bandwidth highly impacts the overall system performance. We implement this policy in **MemEFS**, our locality-agnostic in-memory elastic storage system [6, 5]. Unlike static deployment schemes, MemEFS is able to adapt to its current network infrastructure and to scale dynamically, at runtime, based on the application storage demands. The contributions of this paper are the following:

- We introduce a network-aware mechanism that enables MemEFS to seamlessly access resources located across networks without bandwidth guarantees.

- We revisit the design of MemEFS, our elastic in-memory runtime storage system [13], with a focus on the network-aware data distribution policy.

- We introduce a hot-migration scheme for the MemEFS reconfiguration process that greatly improves the application blocking time;

- We evaluate MemEFS with a variety of real-world and synthetic scientific workflows. We show the efficiency of our design through a set of elastic scaling policies derived from the application storage demand.

Our results show that MemEFS' elasticity improves resource utilization (by up to 65%), while incurring only a modest performance overhead. Our design allows users to trade off resource efficiency for performance. MemEFS' network adaptability mechanism reduces execution time by up to 50%.

This paper is organized as follows. Section 2 sketches the background of our work, while Section 3 introduces the design of MemEFS. Section 4 describes the evaluation results, Section 5 discusses related work and we draw conclusions in Section 6.

2. Background

In previous work [5, 6] we designed MemFS, an in-memory distributed file system for storing the intermediate data of scientific workflows. MemFS is deployed on the nodes on which the application is running and spreads the application data uniformly across these nodes. We showed that, since remote operations have become faster due to increasing network bandwidth and DRAM capacities, a locality-agnostic approach achieves better performance for scientific workflows than state of the art locality-based file systems [6].

MemFS improves the performance of scientific workflows through two important features: (i) it achieves a good load balance for both storage and network traffic, thus avoiding scalability bottlenecks when running data aggregation and partitioning stages; (ii) it maximizes the achieved bandwidth and throughput for read and write operations. MemFS balances the storage load among the nodes by employing a key-value store. Files are striped and each stripe is associated with a key. The node which stores a file stripe is selected by hashing the stripe's key. MemFS uses a modulo hashing scheme, which assigns each stripe to a node in a logical ring, guaranteeing a balanced data distribution. MemFS's file striping mechanism also enables improved read and write throughput by transferring data via parallel streams from multiple nodes.

MemFS is designed to run on tightly coupled, reliable compute resources, such as clusters or supercomputers.

Hence, all-to-all connectivity is ensured and, each node has membership information about all other nodes. This minimizes latency by enabling $O(1)$ file look-up and alleviating the need of additional routing. In environments where nodes join and leave the system at very high rates, such a low complexity would not be possible. As it would be too expensive for every node to keep a full view of the network, only a partial view is used: nodes use routing tables when directing queries, leading to a complexity of $O(\log\ n)$ for look-up operations.

MemFS's implementation is based on three components: (1) a FUSE [14] layer that serves as a POSIX interface to applications and which handles the file striping mechanism; (2) the Libmemcached [15] hashing protocol which determines in $O(1)$ steps which node holds a certain file stripe; and (3) Memcached [16], a fast, in-memory key-value data store.

2.1. Elasticity Requirements

Current in-memory file systems, like MemFS, lack support for adapting their number of nodes to application storage needs, although scientific workflows often have varying storage demands during their runtime. The astronomy workflow, Montage [1], for exmple, is building a mosaic from a set of galaxy images. Montage is composed of a series of stages, each generating different data amounts. Some of Montage's stages are parallel, e.g., composed of thousands of tasks, while others are sequential. Moreover, in Montage, as most of the stages depend only on the data from the previous stage, a part of the runtime data can be deleted by the workflow manager [17], to optimize the total data storage used by the workflow.

Figure 1 shows the variations in utilized data storage during a run of Montage using MemFS (we discard the data that is no longer needed at the end of each stage). If we would have to run Montage on a static number of nodes, we would have to provision for the peak memory utilization (marked as red). Not only would we have to know this peak value before starting Montage, but also, due to varying parallelism levels and data footprint in Montage stages, resources will be left unused while other users might have applications waiting in the cluster's queue.

Adding elasticity to a file system like MemFS involves designing new data distribution and load balancing mechanisms. The key problem is that maintaining the load balance when adding/removing nodes implies moving data among the nodes. As previously shown with MemFS, load balancing can be efficiently achieved by striping files and using a key-value store to store the file stripes. However, adding or removing nodes leads to changing the hash function and thus it invalidates the assignment of stripe keys to nodes.

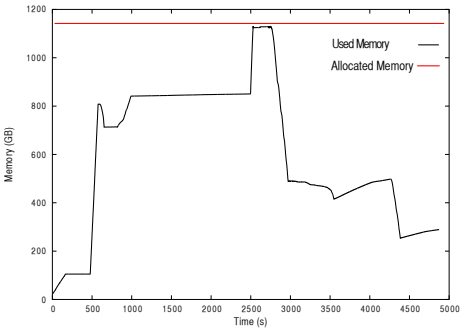


Figure 1: The in-memory storage demand of a scientific workflow during its runtime. Note the difference between the used storage and the allocated amount to satisfy the peak demand.

The motivation for this behaviour stems from the hashing mechanism inherent design and purpose: hash functions map objects (stripe keys) to a *fixed number* of containers (nodes). A *good* hash function should achieve a load-balanced distribution of objects to containers. When the number of containers changes, the hash function needs to change (to take into account the newly added/removed containers), as otherwise, the mapping of objects to containers will be invalidated (i.e., objects will not be found).

As nodes are added or removed, not only the hash function needs to be changed, but also objects need to be migrated to the containers where the new hashing function would map them.

The movement of data during reconfigurations degrades the application performance, as application I/O has to be postponed until the reconfiguration has finished. To reduce the application performance degradation, the amount of data migrated among nodes needs to be minimized.

In this paper, we leverage the key results of MemFS to design MemEFS, an *elastic in-memory distributed file system*. MemEFS *improves* MemFS through: (i) efficient mechanisms to store and re-distribute the data when adding/removing nodes; (ii) elastic scaling policies to adapt the number of nodes to variations in application storage demand.

2.2. Network adaptation requirements

With a highly distributed design, file systems like MemFS are not robust to network variability. While in a private infrastructure setup this aspect is downplayed by the intrinsic performance isolation of the execution

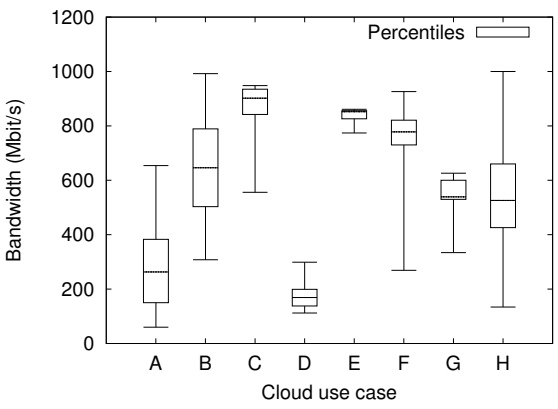


Figure 2: Bandwidth distributions for A-H cloud use cases. 1st-25th-50th-75th-99th percentiles. Data points courtesy of authors of [10].

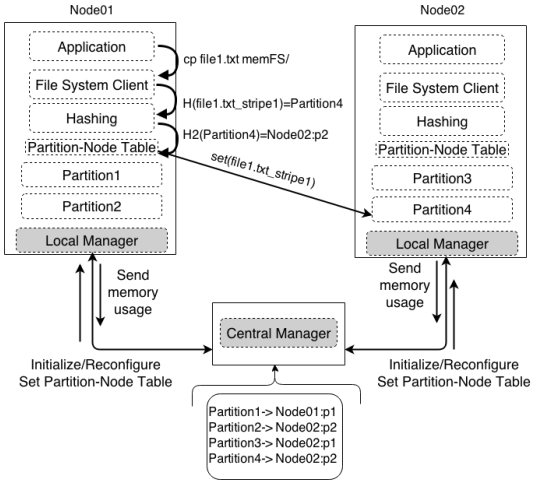


Figure 3: Architecture of MemEFS.

environment, the situation is radically different in public clouds. Figure 2 plots the results of an exhaustive study [10] which showed that the network performance is highly variable in public clouds. Such an imbalance in the bandwidth observed by MemFS nodes would lead to a general slowdown of the system, as faster connected nodes will be reduced to the speed of the slower connected ones.

Thus, adding network adaptability to a file system like MemFS also affects the design of its data distribution and load balancing mechanisms. In this paper, *we introduce a network-aware data distribution policy in the design of MemEFS*.

3. MemEFS Design and Implementation

Figure 3 shows an overview of MemEFS, consisting of worker nodes and a Central Manager (CM). The CM gathers worker node statistics, takes reconfiguration decisions based on these statistics and orchestrates the reconfigurations. The worker nodes run a File System Client, a Local Manager and the application processes. The Local Manager monitors the node's resource statistics, e.g., memory utilization, and sends it to the CM. The CM may run on a separate node, but it can also reside on a worker node without affecting the worker's performance. An important remark is that in MemEFS the worker nodes have a dual role: they participate in both storing data and running applications.

When worker nodes are added or removed, MemEFS needs to move data to maintain the load balance. To minimize the data movement, and thus maintain application performance, MemEFS uses a consistent hashing scheme [18]. This guarantees that in a system that holds K objects on N nodes, when a node is added, at most $O(K/N)$ objects need to be rehashed.

MemEFS implements consistent hashing through a *two-layer hashing scheme* that maps file stripes to *partitions* and then partitions to nodes. We assume it is preferable to organize data in a manner that permits moving small numbers of large objects (partitions) rather than large numbers of small objects (file stripes). Better performance is achieved when transferring larger objects since fewer data transfers are needed, and hence we minimize the latency and maximize the bandwidth utilization. With this argument in mind, each node holds multiple partitions, such that, when reconfiguring the file system, we migrate partitions, and thus avoid rehashing the file stripes.

Throughout the application runtime, the number of partitions is constant. The total number of partitions sets the upper bound on the number of nodes to which the elastic distributed file system can scale out to: there cannot be more nodes than partitions. Therefore, the size of each partition is limited by the memory capacity of its host node. Thus, when running on a small number of nodes with many partitions, the partition size will be small. When scaling out to a larger number of nodes, a subset of the partitions will be migrated to the newly added nodes, allowing all the partitions to grow in size. The growing or shrinking of partitions is dependent on the application behaviour: if the application writes more data, the number of file stripes per partition increases; conversely, if the application removes data, file stripes are deleted from the partition.

3.1. Two-Layer Hashing Scheme

To store file stripes, MemEFS uses the two layer hashing scheme as follows. The mapping of stripes to partitions is achieved using the xxhash [19] algorithm. This algorithm hashes an input string to a 64-bit number. We have chosen this non-cryptographic algorithm because it is optimized for 64-bit CPUs, yielding up to 13GB/s throughput [19] and outperforming other hashing algorithms like SHA1 [20] or MD5 [21] by two orders of magnitude. This is important for MemEFS since a fast hashing scheme reduces the latency of looking up a file stripe. To determine the mapping of a stripe to a partition, we hash the *file path* and the *stripe number*, obtaining a 64-bit integer. We then use a modulo (circular) scheme to determine which partition holds the file stripe.

The mapping of partitions to nodes is kept in a table, called the *Partition-Node* table. This table is stored on each node and it is updated by the CM at each reconfiguration. Figure 3 shows the steps required to read or write a file stripe: MemEFS first determines the id of the partition responsible for the file stripe, then the node responsible for the partition, and then the query is sent directly to that node. Thus, MemEFS achieves $O(1)$ look-up for storing or retrieving file stripes.

3.2. Load Balancing

MemEFS computes the number of partitions each node stores after each reconfiguration by adapting the Y0 algorithm proposed in [22]. This algorithm computes the mapping of partitions to nodes. However, the mapping of file stripes to partitions is always constant: once a file stripe has been assigned to a partition, it will always reside there, even though the partition may be migrated to different nodes.

Y0 improves on the Chord [23] DHT and achieves good load balance, even when nodes are heterogeneous in terms of storage. The load imbalance in Y0 was shown to be a constant factor of at most 3.6, while DHTs usually generate a load imbalance in the order of $O(\log\ N)$.

The core idea of Y0 is as follows. Considering there are n heterogeneous nodes, to achieve load balance, each node v should own a fair share $share(v)$ of the storage capacity. This share is computed as the ratio between the fraction of the storage assigned to node v (f_v) and the fraction of the total normalized capacity belonging to node v (c_v/n):

$$share(v) = \frac{f_v}{c_v/n} \tag{1}$$

The normalized capacity of node v is generically defined as:

$$c_v = \frac{Capacity(v)}{\frac{\sum_{u \in Nodes} Capacity(u)}{n}} \tag{2}$$

and it is easy to verify that $\sum_{v \in Nodes} c_v = n$.

The system is load balanced when each node's share is equal to 1 and therefore the f_v values will determine the system's load balance. The authors show that, when each node holds $2 \log\ n$ partitions per capacity unit, the system's imbalance factor is at most 3.6.

To achieve load balance in MemEFS, we adapt the Y0's computation of nodes' shares by defining the capacity of each node v in terms of memory:

$$c_v = \frac{Memory(v) \times n}{\sum_{u \in Nodes} Memory(u)} \tag{3}$$

We also define the fraction of the storage assigned to node v in terms of the number of partitions of node v (P_v) compared to the total number of partitions in the system, $P = \sum_{v \in Nodes} P_v$. Then, the share of node v becomes:

$$share(v) = \frac{P_v/P}{c_v/n} \tag{4}$$

If there are n nodes initially in MemEFS, then there are at most $2n\ \log\ n$ partitions in total. Since P_v cannot be smaller 1, it follows that MemEFS would not be able to add more than $2n\ \log\ n$ nodes to the system when scaling out.

It has been shown [22] that a higher number of partitions further reduces the imbalance factor. However, in the case of Y0, a larger number of partitions per capacity unit adds significant overhead to look-up operations due to the size increase in the finger table. MemEFS is not affected by this, as it delivers $O(1)$ look-up operations. Therefore, when defining the number of partitions of a node, P_v, instead of using $2 \log\ n$ partitions per capacity unit, we introduce a *scaling* constant β, such that:

$$P_v = c_v \beta \log n \tag{5}$$

Through this scaling factor MemEFS gives users more control in defining how much the system could scale out, as the number of partitions determines the maximum number of nodes the system can scale out to. The partitions are variable in size: when there are many on a machine, they are smaller, but when scaling out, the number of partitions on a machine decreases and they are allowed to grow in size.

3.3. Network Awareness

As typical private clusters have homogeneous networks, nodes of the same type receive the same number of MemEFS partitions. In public clouds, when dealing with highly heterogeneous links, we assume that it is beneficial to distribute MemEFS partitions in such a way that nodes with more bandwidth host more partitions.

In a cloud setup, we start by measuring the available bandwidth of the MemEFS virtual machines. Next, we adapt again the Y0 algorithm to compute the numbers of partitions per virtual machine according to the bandwidth capacity. Equation 3 becomes:

$$c_v = \frac{Bandwidth(v) \times n}{\sum_{u \in Nodes} Bandwidth(u)} \quad (6)$$

The network-aware Y0 algorithm uses the same principles described previously. In the cloud setup, only the capacities of the nodes are now computed based on the observed bandwidth of the nodes.

During the runtime of an application, MemEFS monitors the available bandwidth of its nodes. If the bandwidth capacity of a node changes, the Y0 algorithm is re-run and the partition-to-node mapping is updated accordingly. Even though migrating data from a node with low bandwidth may reduce performance momentarily, by applying this technique we optimize for the *long-term behaviour* of the system, as the application execution time is a-priori unknown.

It is important to note that the Y0 algorithm need not consider the *absolute* (maximum) bandwidth of a node, but actually its *observed* bandwidth, i.e., the amount of network traffic generated by the application. This is sufficient, as generally workflow tasks are uniform in their resource requirements: tasks from the same stage exhibit similar amounts of I/O, memory, CPU operations. Therefore, in stages that exhibit low I/O, the network bandwidth will be under-utilized in all nodes. Furthermore, this mechanism enables a simple procedure to decide when a node's bandwidth is maximized.

When the application increases its network traffic, the nodes that exhibit lower bandwidth will not exhibit a utilization past their actual physical limit. Conversely, in nodes with higher bandwidth, the increase in utilization will be higher. Therefore, in periods of high network utilization, by analyzing the amount of increase in observed bandwidth in each node, we can decide which nodes are bandwidth-contended.

3.4. Initialization and Reconfiguration

Next, we discuss the steps required by MemEFS to initialize and reconfigure itself.

3.4.1. File System Initialization

MemEFS starts with a node running the *Central Manager* (CM). The CM process takes as input the number of initial worker nodes and β (all experiments in this paper use $\beta = 4$). Then, the CM starts the worker nodes. Based on the underlying environment (cluster or cloud), the CM creates the required numbers of partitions on the worker nodes using the corresponding load balancing scheme as introduced in Section 3.2 and 3.3, respectively. Then, it creates the Partition-Node table and broadcasts it to all worker nodes. When the worker nodes receive this table, they also mount the File System Client.

3.4.2. File System Reconfiguration

When an application is running, the Central Manager queries all worker nodes for their memory utilization. The time interval at which the queries are done is configurable, with a default value of one second. Based on memory utilization information, the file system reconfigures itself automatically during application runtime by adding or removing workers (nodes or virtual machines).

To start new nodes, in a cluster setup, the CM interacts with the cluster queueing system. Conversely, in a public cloud setup, the CM utilizes the cloud API, or a library such as Apache Libcloud [24].

After the new workers have been started, or before existing workers are removed, the CM determines how many partitions have to be migrated and where, and rebalances the workers. Node removal is possible when the remaining set of nodes have enough memory to store all the data. The CM removes a part of the current nodes if the memory utilization over a certain time span, e.g., 45 seconds, does not increase, and is below a given threshold.

If (a part of) the newly added nodes are running in a public cloud, the CM monitors their achieved bandwidth over a (configurable) period of time. If there is significant variability, the network-aware Y0 algorithm is run, and the partitions are redistributed accordingly.

Because during reconfigurations the partition-to-node mapping changes, any reads or writes issued by the application would be invalid and thus, before reconfiguration starts, the application I/O operations are suspended. The I/O operations are resumed only after the reconfiguration finishes, i.e., the data is migrated and each node has the new partition-to-node mapping.

3.4.3. Elastic Scaling Policies

We designed several *elastic scaling* policies. The policies can scale out or in the number of storage (also

used for computation) nodes. By *scaling out* we denote the process of adding nodes to the system. Conversely, by *scaling in* we denote the process of removing nodes from the system. The various combinations of these policies lead to trade-offs between resource utilization efficiency and workflow execution speed.

Our working assumption is that the system does not have any prior knowledge of the applications. Moreover, scientific workflows are composed of sets of highly heterogeneous tasks. The heterogeneity lies not only in task runtimes, but also in CPU utilization, I/O utilization and patterns, and output file sizes. Therefore, to improve resource utilization, the only viable approach is to let the system be modeled by the application data demand. In this way, we make sure that the application uses only as many resources as needed.

For scaling out, we define three policies, ranging from a conservative to an aggressive approach. If the *scaling out* policy is more conservative, i.e., scales with small number of nodes, the system would also benefit from less compute resources, since in MemEFS the storage is colocated with the compute nodes. Hence, the more conservative the *scaling out* policy is, the higher the application slowdown. These policies are summarized as follows:

- **CSO** - Conservative Scale Out: assuming the system starts with N nodes, we always scale out by $\frac{N}{2}$ nodes when the total system utilization grows higher than 95%.
- **NSO** - Neutral Scale Out: assuming the system starts with N nodes, we always scale out by N nodes when the total system utilization grows higher than 95%.
- **ASO** - Aggressive Scale Out: we always double the current number of system nodes when the total system utilization grows higher than 95%.

For scaling in, we define two policies: conservative and aggressive. The more aggressive the *scaling in* policy is, the higher the application slowdown. These policies are summarized as follows:

- **CSI** - Conservative Scale In: when the total system utilization drops below 75%, we remove 25% of the nodes.
- **ASI** - Aggressive Scale In: when the total system utilization drops below 50%, we remove 50% of the nodes.

The amount of data migrated depends on the nodes that are added or removed. To be more precise, the amount of migrated data is proportional to the capacity units measured in the Y0 algorithm - either memory capacity, or network bandwidth.

In a setup with homogeneous network links, each node participates equally when storing the data. As in our approach nodes are both storage and compute, when $X\%$ new capacity units are added, $X\%$ data is migrated to the newly added nodes. Also, according to previous research [5], there is no need to minimize data migration further than achieving balanced storage, as balance is key for performance. Furthermore, the Y0 consistent hashing scheme ensures that the minimal amount of data will be migrated, while preserving load balance.

With heterogeneous network capacities, our system achieves a load balance proportional to the network link capacities such that the overall computation is not slowed down by the network imbalance.

When running in a cloud setup we choose nodes which have similar characteristics (CPU, memory). In typical commercial clouds, the amount of resources are also correlated with the network capacities[1]. As such, instances with large CPU and memory have higher available bandwidth. The reciprocal is also valid. Therefore, in cloud setups, we only have to cope with the bandwidth variability (as all other resources are similar), task which is performed by the network adaptation mechanism.

In such setups, when node addition/removal is performed, we compute the number of partitions per node based on bandwidth capacities. If the newly added nodes do not have enough bandwidth to free up some nodes which have their memory filled up, we continue adding nodes until enough partitions are redistributed and the system can continue operation. A node would have to exhibit several orders of magnitude lower bandwidth (variability much larger than the cases presented in Figure 2) for such situations to occur, which is highly improbable in the cluster and cloud setups discussed in this paper.

3.5. Implementation

We next describe several implementation decisions for MemEFS. We discuss the data store choice, the communication protocol between the Central Manager and the workers, and the implementation of the file-system client. Finally, we discuss how MemEFS could implement fault tolerance.

[1] `https://aws.amazon.com/ec2/instance-types/`

3.5.1. Data Store

To store file stripes, MemEFS relies on existing key-value data stores. However, an important feature required by MemEFS is to allow data partitioning and partition migration among nodes. Originally, MemFS used Memcached for storing data in memory. Because in MemEFS we use the concept of partitions, and each node might store multiple partitions, we could have implemented the partitions as Memcached databases. However, when reconfiguring the file system, there was no mechanism for migrating a Memcached database between nodes. Therefore, in MemEFS we have opted to use Redis as the key-value data store [25]; each partition is a database managed by a Redis process.

Redis has several mechanisms to migrate databases between nodes: (i) cold migration, i.e., dump the database to a file and transfer the file to the new node; (ii) master-slave replication (hot migration): we can obtain a copy of the initial Redis process by setting the new Redis process as its slave; after the replication has finished, we can simply kill the original Redis process; (iii) record a log to disk with all operations that have altered the database; for migration the log can be copied to the new node and replayed by the new Redis process.

MemEFS uses two mechanisms to move partitions between nodes during reconfigurations:

1. **Cold Migration:** When the reconfiguration starts, the application processes are stopped. Then, the target Redis servers dump their databases to disk. Afterwards, the dump is copied over the network to the newly added node and the Redis process is restarted using the database dump. Only after all this tedious process has finished, the application can be restarted.
2. **Hot Migration:** The newly added nodes are started in advance with one Redis server running as a *slave* that replicates the Redis server contents of the original node. When the replication is close to being finished (i.e. at 90%), the application is blocked until the slave is an exact replica of the master. After this finishes, the Redis master is killed, the partition table updated to point to the newly added Redis replica, and the application processes are finally un-blocked.

While in the original implementation [13] MemEFS used only a cold migration scheme, in this paper we also implement the hot migration scheme. This scheme greatly improves the running times of the applications by employing shorter blocking times during the reconfiguration process. The log-based mechanism of Redis replication is not suitable, as it would increase the blocking times even further than cold migration. This is because usually log files are larger than the database dumps, and replaying the log file might be more time consuming than loading the dump file.

3.5.2. Communication between Central Manager and Workers

Each worker node runs a Local Manager (LM) process that communicates with the Central Manager. Figure 4 shows the communication protocol. The LM process runs three threads. One thread measures the local memory and bandwidth utilization at regular time intervals; the time interval is configurable with a default value of one second. The memory utilization is monitored by checking how much memory is allocated by the redis processes. To check the bandwidth utilization, the LM constantly polls the */proc/net/dev* file exposed by the Linux Kernel and stores the amount of data that had passed through the network interface. Assuming 1 second measuring intervals, the LM then computes the bandwidth achieved by the node in the previous second.

A second thread sends the local memory and bandwidth utilization information to the CM whenever this information is requested. A third thread listens for reconfiguration messages.

When the Local Manager receives the message that the reconfiguration should start, it first sends a signal to the file system client (FS) to suspend application I/O operations. After the FS acknowledges that the application's I/O operations have been suspended, the LM sends an acknowledgment to the CM signaling that the reconfiguration can take place. When the reconfiguration has finished, the CM sends a message with the new Partition-Node table to the LM. When the LM receives this message, it sends another signal to the FS, announcing that it should now reload its Partition-Node table and then safely resume the application's I/O operations.

3.5.3. File System Client

The MemEFS file system client is implemented as a FUSE module which communicates with Redis using the `hiredis` communication library [26]. We have extended the implementation of MemFS's file system client (FS) to support elasticity and suspend/resume actions for the application's I/O operations during reconfigurations. When receiving the reconfiguration signal, the FS waits until the current `read()`, `write()` or other application request finishes, then blocks all other incoming application requests, and sends back an acknowledgment. The blocking is implemented in the request handling code as a wait on a semaphore. When

the FS receives the signal to resume the application operations, it first recomputes the Partition-Node table and then resumes the application's I/O operations - by incrementing the semaphore.

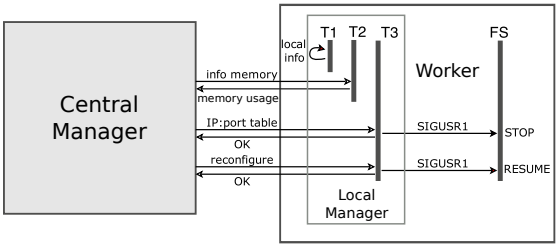


Figure 4: Communication between the Central Manager and a worker.

3.5.4. Fault-Tolerance

MemEFS can be configured as fault-tolerant and persistent, however fault-tolerance is outside the scope of this paper. Making MemEFS fault-tolerant involves providing data fault-tolerance and high availability for the Central Manager. Data fault-tolerance can be provided by Redis, which achieves fault-tolerance through replication. We believe that replication is not a good strategy for in-memory runtime file systems because it largely increases memory usage, while also slowing down the writing operations. The Central Manager can be made highly available by leveraging state-of-the-art solutions [27]. We defer to future work studying the different strategies for fault-tolerance, such as erasure coding for data and state machine replication for the Central Manager.

4. Evaluation

We evaluate MemEFS from two perspectives: elasticity and network adaptability using typical scientific workloads. In terms of elasticity, we show how MemEFS can control the trade-off between resource utilization efficiency and application performance. In this scenario, we use a suite of typical scientific workflows and a set of elastic policies in a cluster setup.

In terms of network adaptability, we again show how MemEFS can control the trade-off between resource utilization efficiency and application performance. In this scenario, we use the same suite of typical scientific workflows in a cloud setup.

Raw performance metrics (bandwidth, latency) are presented in our previous work [5], where we extensively study the performance and scalability of MemFS. In this paper we only focus on MemEFS' elasticity and network adaptability, and show how real-world applications can benefit from it.

Table 1: Workflows Characteristics

Application	# Tasks	Input Size	Peak Storage
Montage	139918	51GB	1TB
BLAST	41472	57GB	550GB
Broadband	1080	6.8GB	700GB
Cybershake	81721	230GB	870GB

4.1. Evaluation Applications

As evaluation applications we used two real-world and two synthetic scientific workflows. Table 1 describes the characteristics of these workflows in terms of total number of tasks, size of input data and peak value of stored data during their runtime. The two real-world workflows are Montage [1] and BLAST [2] - their source code and input data are available online. Montage is an astronomy application that builds a mosaic from a set of input images of a galaxy. The size of the application depends on the number of input images. Montage is composed of multiple stages in which a different binary is run for various image operations, e.g., processing, aggregation, partitioning of results. BLAST is a bioinformatics application that searches for specific nucleotide sequences in a database. Like Montage, BLAST is also composed of multiple stages involving partitioning, processing and aggregation of data. For Montage, we used a 20×20 mosaic centered on the M17 galaxy, while for BLAST we used the NCBI *nt* database.

The two synthetic scientific workflows are Broadband [28] and Cybershake [28]. Because their code and input data are not openly accessible, we generated two synthetic workflows using publicly available execution traces of their real-world counterparts [29, 30]. We selected these two from the public repository of workflows [29, 30] since these generated the largest data amounts. To generate the synthetic workflows, we used the Application Skeletons framework [31]. This framework allows the user to specify the data usage patterns, task runtimes and task dependencies.

4.2. Elasticity Evaluation

To show how MemEFS scales elastically with the application storage demands, we ran all 4 workflows under all valid elastic scaling policy mixes in a cluster setup. We use the set of elastic scaling policies described in Section 3.4.3. We also introduce two performability metrics, the resource utilization improvement and the

performance overhead, to help compare the efficiency of the various elastic scaling policies. Next, we evaluate the degree of storage load balancing achieved by these scaling policies.

4.2.1. Cluster Setup

The elasticity experiments were executed on the DAS4 multi-cluster system [32]. Each compute node is equipped with a dual-quad-core Intel E5620 2.4 GHz CPUs and 24GB memory. The nodes are connected by a commodity 1Gb/s Ethernet and a premium Quad Data Rate (QDR) InfiniBand providing a theoretical peak bandwidth of 32Gb/s. For our experiments, we chose to use the IP over InfiniBand (IPoIB) interface of the latter, which delivers approximately 1GB/s bandwidth. For all experiments, out of the 24GB node memory, we allocated 20GB to MemEFS and left 4GB for the operating system. In our setup, the compute nodes, which run the application tasks, also act as storage nodes for MemEFS. Thus, when scaling MemEFS, the application also scales. In all experiments, the user has to provide the initial number of nodes, N, on which MemEFS and the application are deployed. We consider that this number can be easily computed, for example by using the size of the input data. Using the input data size to determine the initial number of nodes, in our experiments, Montage starts on 16 nodes, BLAST on 8 nodes, Cybershake on 32 nodes, and Broadband on 8 nodes.

4.2.2. MemEFS Performability

For applications where intermediary data may be discarded since it is no longer needed, we evaluate different mixes of scaling out and scaling in policies. For applications that only exhibit an increasing data usage, we evaluate MemEFS only under scaling out policies. In this section, we only present results using the *cold migration scheme*.

To illustrate the behavior of the policies, Figure 5 shows the resource utilization versus allocated storage during the runtime of Montage for all the six possible scaling policy combinations. As opposed to the increasing and shrinking data footprint of Montage, BLAST' data footprint only increases. Therefore, we evaluate BLAST only under the three scaling out policies. Figure 6 shows the three elastic policies behavior when running BLAST.

For brevity, we do not present the elastic scaling behavior of the Cybershake and Broadband workflows, as their scaling patterns are similar to those of Montage and BLAST, respectively. However, we report their performability to show how the scaling policies affect the resource utilization efficiency and the application performance.

To evaluate the performability of MemEFS, we executed the workflows on two deployment schemes: static and elastic. In the *static* deployment scheme, MemEFS provisions enough nodes to store all the data generated by the application. In the *elastic* deployment scheme, MemEFS starts by provisioning enough nodes to copy the application input data and afterwards it uses the previously discussed scaling policies. We repeated each experiment 4 times and report the average.

We use the following performability metrics to assess the quality of the elastic policies:

- **Resource Usage Improvement:** represents the amount of resources (memory, nodes) wasted for an elastic run compared to the amount of resources wasted for the static run. The resource usage improvement is defined as:

$$RUI = \frac{W(static) - W(elastic)}{W(static)}.$$

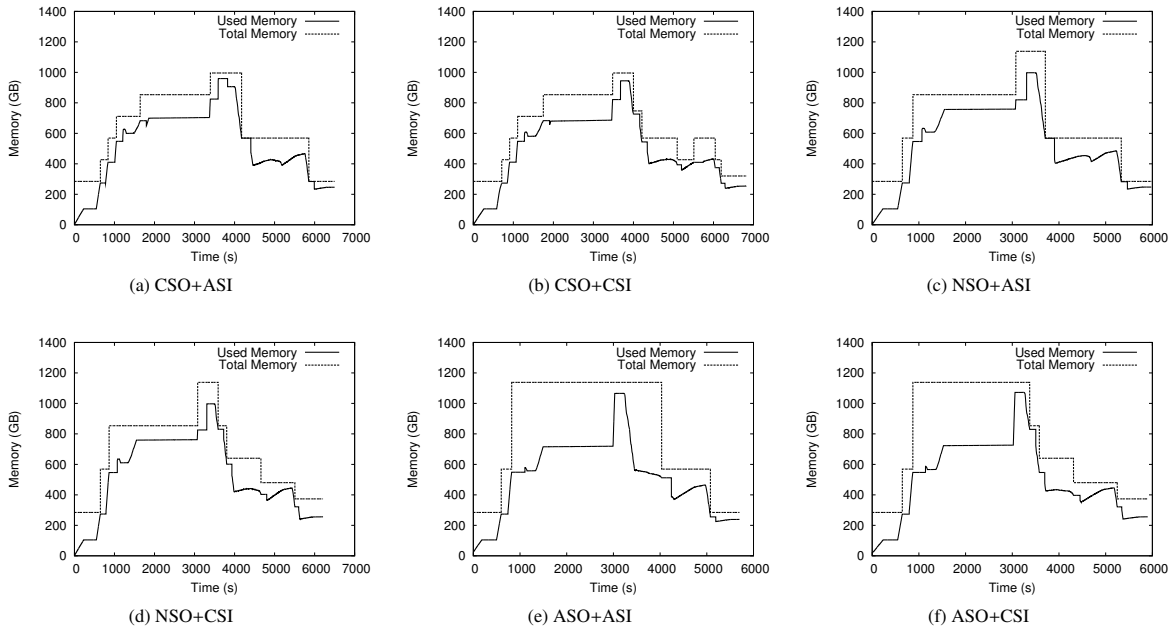
  $W(s)$ represents the amount of wasted resources in a deployment scheme s and is defined as the amount of allocated resources unused by the application:

$$W(s) = A(s) - U(s),$$

  where $A(s)$ is the amount of allocated resources, and $U(s)$ is the amount of used resources. The higher the value of this metric, the better the policy.

- **Performance Overhead:** represents how much slower is an elastic run compared to the static run. This overhead captures the impact of an elastic run on the application performance: reconfiguration overheads and less compute capacity. The lower the value of this metric, the better the policy.

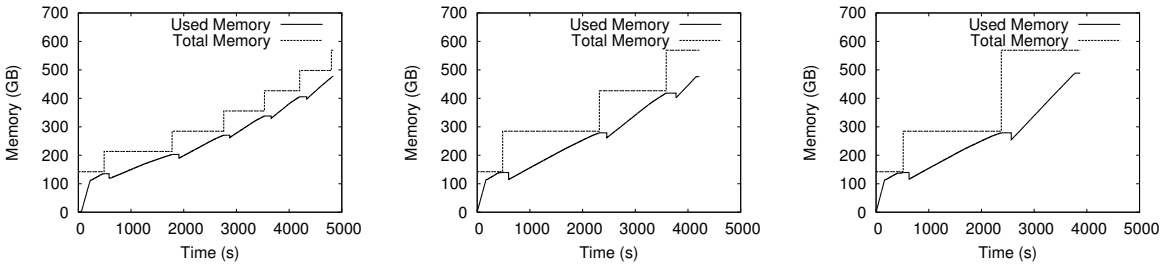
Figure 7a presents the results of the elastic policies for running Montage. Plot bars are sorted according to policy aggressivity. During the runtime of Montage, intermediate data that is no longer needed is discarded and, thus, we evaluate all possible combinations of our proposed policies. As expected, when increasing the aggressivity of our policies, the resource usage improvement decreases together with the performance overhead. This is explained by the fact that more aggressive policies use more workers, thus achieving a better application speedup. For Montage, the RUI varies between 31.9% (**ASO+ASI**) and 65.7% (**CSO+CSI**), while the performance overhead varies between 14.4%



(a) CSO+ASI (b) CSO+CSI (c) NSO+ASI

(d) NSO+CSI (e) ASO+ASI (f) ASO+CSI

Figure 5: The behavior of elastic scaling policies for Montage.



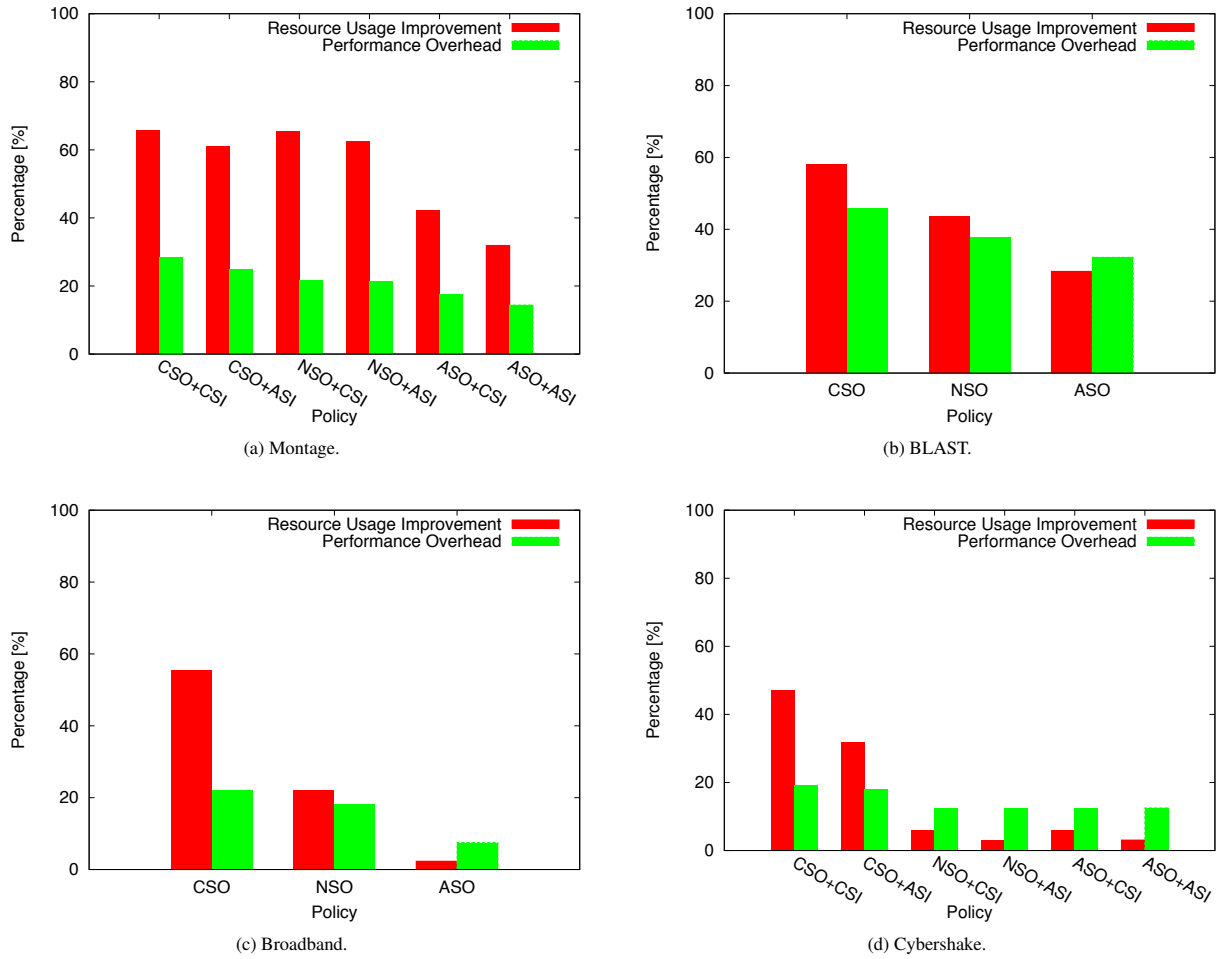
(a) CSO (b) NSO (c) ASO

Figure 6: The behavior of elastic scaling policies for BLAST.

(**ASO+ASI**) and 28.5% (**CSO+CSI**). The large difference between the two metrics is given by Montage's design: large parallel stages are mixed with long sequential stages (synchronization points represented by data aggregation/partitioning stages). During the sequential stages, our policies are able to largely improve the resource utilization without incurring performance overhead.

Figure 7b shows the policies evaluation results when running BLAST. For BLAST, no intermediary data may be discarded, thus we only evaluate the scaling out policies. As expected, when the policy aggressivity increases, both evaluation metrics show a decreasing trend, with the performance overhead being proportional to the resource utilization improvement. In this scenario, the RUI ranges from 28.3% (**ASO**) to 58% (**CSO**), while the performance overhead ranges from 32.2% (**ASO**) to 45.7% (**CSO**). As opposed to Montage, BLAST does not have sequential stages and running with less worker nodes will slow down the execution.

Figure 7c shows the policies evaluation results for the synthetic Broadband workflow. Similarly to BLAST, intermediary data may not be discarded, thus we could not evaluate the scaling in policies. Again, both performability metrics decrease as the policy aggressivity increases. The resource usage improvement varies between 2.3% (**ASO**) and 55.6% (**CSO**) and the performance overhead varies between 7.5% (**ASO**) and 22.2% (**CSO**). Interestingly, for Broadband, the **CSO** policy



(a) Montage.

(b) BLAST.

(c) Broadband.

(d) Cybershake.

Figure 7: Impact of scaling policies on resource utilization efficiency and application performance for various workflows (lower is better for performance overhead; higher is better for resource usage improvement).

outperforms all others, with a 55.6% resource usage improvement while incurring only a 22.2% performance overhead. This is explained by the Broadband workflow structure [30]: in the first stages, the workflow is less parallel, while the final stages exhibit a large degree of parallelism. Thus, the CSO policy largely decreases the resource utilization in the first stages without considerably slowing down the application.

Figure 7d shows the policies evaluation results for the synthetic Cybershake workflow. Because Cybershake needed a larger storage for its input, we started this workflow with more nodes than the previous workflows, i.e., 32 nodes. Because our cluster is limited to only 64 nodes, the **NSO** and **ASO** policies produce the same values for both the resource utilization improvement and the performance overhead metrics. In Cybershake, intermediary data may be discarded and, thus, we evaluate all possible combinations of our proposed policies.

Again, when the aggressivity of the policy increases, the performability metrics show a decreasing trend. Similarly to Broadband, the conservative policy outperforms all others. The **CSO+CSI** saves 47.04% resources, while increasing by only 19.9% the total runtime. The explanation for this behavior is similar to the Broadband case: Cybershake exhibits less achievable parallelism in the first workflow stages and the CSO policy is therefore able to save more resources without increasing too much the runtime.

Furthermore, for each application we have chosen the best performing scaling policy and compared it to the static version in terms of the total amount of acquired

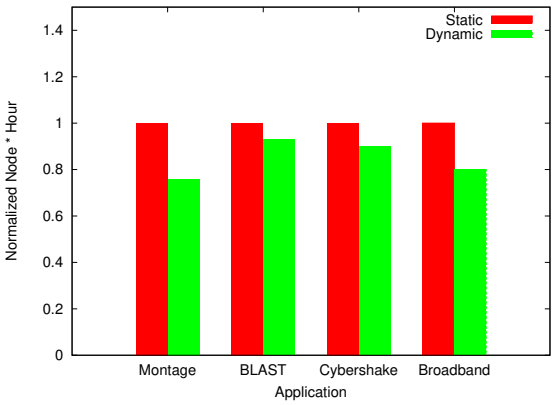


Figure 8: Normalized Node Hours (lower is better).

resources (node-hours). Figure 8 shows the savings in normalized node-hours generated by elastically running the applications. The node-hours reduction ranges from 8%-10% (BLAST and Cybershake) to 18%-22% (Broadband and Montage). We argue that this is an important result since these applications are usually run many times by scientists, typically in conjunction with parameter sweeps, to explore the entire parameter space. Hence, our approach would greatly reduce electricity bills in private clusters and/or operational bills in public cloud scenarios.

4.2.3. Hot Migration

To show the added benefits of *hot migration*, we repeated the previous Montage and BLAST experiments. During these runs, for all scaling policies, we enabled the hot migration scheme.

Figure 9 compares the time taken to perform hot and cold migration when running BLAST with the three scaling policies. For brevity, we omit the detailed results of individual policies when running Montage. We notice that the hot migration duration is nearly constant, and always below 10 seconds. In contrast, the cold migration takes longer, and depends on the size of the migrated data. Migration time corresponds to application blocking time, therefore, the application runtime is largely decreased using our newly designed hot migration scheme.

To show this, in Figure 10, we plot the runtime overhead (application slowdown) introduced by two reconfiguration schemes when running BLAST (Figure 10a) and Montage (Figure 10b). We noticed that for the cold migration scheme, when running BLAST, the slowdown is at most 13%. In addition, for Montage, the slowdown varies between 13% and 24%. In contrast, in the case of hot migration, for both applications, the slowdown is actually negligible: below 5% for Montage and below 2% for BLAST.

Although the application runtime is highly improved when using hot migration, the normalized node-hours results (see Figure 8) still hold. This is because the migration process takes approximately the same amount of time for both hot and cold migration. In the case of hot migration, this migration time is simply *hidden* from the application, as the newly added nodes are deployed in advance to start replicating data. Also, the same amount of data is migrated, but asynchronously to the application.

4.2.4. Storage Load Balancing

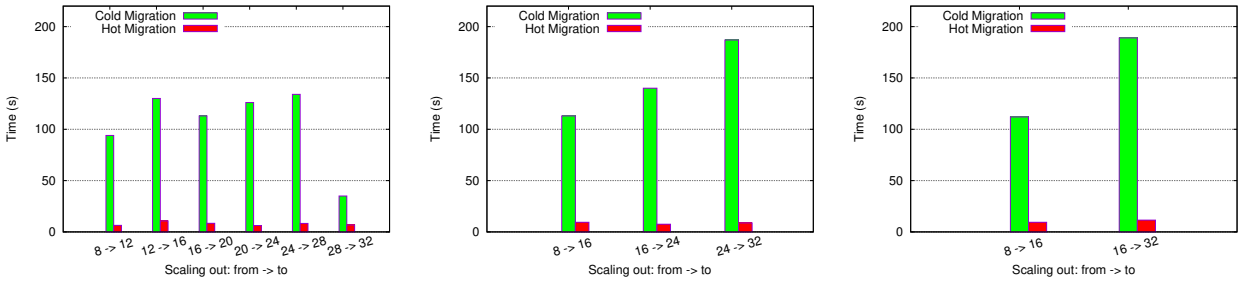
Figure 11 shows an in-depth analysis of MemEFS storage load balance during an elastic run. For this experiment, we selected a Montage execution, using a smaller input compared to previous experiments, and the **NSO+ASI** elastic scaling policy. The experiment starts on 8 nodes, reaches a peak of 24 used nodes and finishes on 12 nodes - the system scales out twice and scales in only once. We measured the memory utilization of all the nodes in the system at each second. To measure the load imbalance, we computed the average and standard deviation of the memory utilization values.

The peaks and the valleys followed by a decrease or, respectively, increase in memory utilization represent the behavior of scaling out/in. A decrease after a peak means that the system has scaled out and some of the partitions have been moved to other nodes leading to an overall decrease in utilization for all the nodes. A sudden increase after a low point on the graph represents the scaling in behavior. After the utilization drops below the scaling in threshold, some partitions are migrated and a part of the nodes removed from the system. Thus, the overall utilization in the remaining nodes increases.

Figure 11 shows the average node memory utilization and the standard deviation for each 1 second time interval. We notice that the difference between the average and standard deviation is at most 17%. This imbalance is given by the partition granularity: the initial number of partitions does not divide evenly to the number of nodes after two reconfigurations. Hence, a subset of the nodes holds more partitions than the others and the system exhibits a small load imbalance (17%).

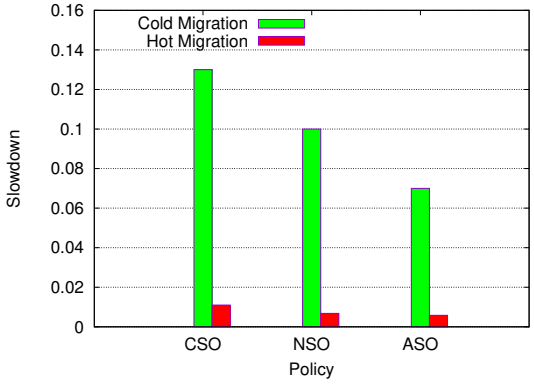
4.3. Network Adaptability Evaluation

To show the efficiency of MemEFS adapting to cloud-like network conditions, we first assessed the optimization potential of the network-adapted MemEFS and then

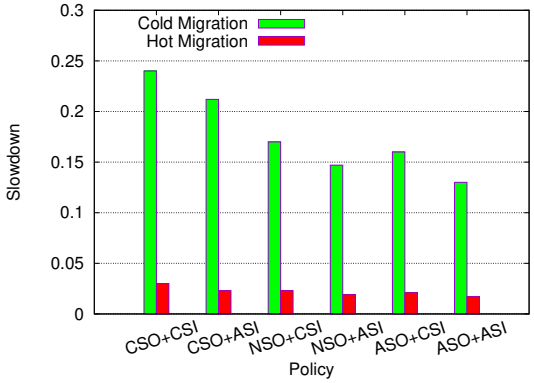


(a) CSO. (b) NSO. (c) ASO.

Figure 9: Hot- vs. Cold-Migration when running BLAST. Scaling out from 8 to 32 nodes with different policies, showing each individual scale out.



(a) BLAST.



(b) Montage.

Figure 10: Hot- vs. Cold-Migration application overhead when running BLAST and Montage.

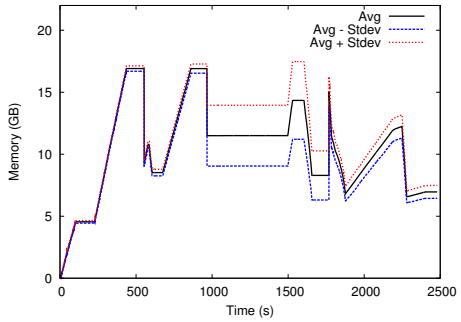


Figure 11: Load balancing analysis for a run of Montage.

studied its behavior on real-world workflows. As network variability would affect the overall application runtime, we evaluate in a cloud setup the performance of MemEFS in terms of the application runtime.

4.3.1. *Cloud Experimental Setup*

In [10], Ballani et al. present the network variability in eight real-world cloud data centers. We use these eight use cases to emulate the real-world network bandwidth variability in our controlled Open Nebula [33] environment installed on DAS4 [32]. We chose this approach over directly using a public cloud to create a controlled environment.

To emulate these eight use cases [10], we use *the hose model* [34] to control bandwidth in our Open Nebula deployment. This model is a simple virtual network overlay able to limit VM-to-VM traffic according to the user specification. For our experiments, we use 32 VMs that each have 8 cores and 20GB of memory. Their allotted bandwidth capabilities follow the distributions reported in [10], shown in Figure 2.

4.3.2. *Network Adaptability Experiments*

We first study the optimization potential of the network-adapted MemEFS by running an I/O-intensive microbenchmark on each setup. The workload is composed of 1000 tasks, each writing 100MB of data to MemEFS. We compared the performance achieved by the network-adapted MemEFS to the network-agnostic

Table 2: Workflow Characteristics

Application	# Tasks	Input Size	Peak Storage
Montage	39472	13GB	320GB
BLAST	3072	57GB	192GB

version, that distributes partitions evenly across nodes. Figure 12 shows the evaluation results.

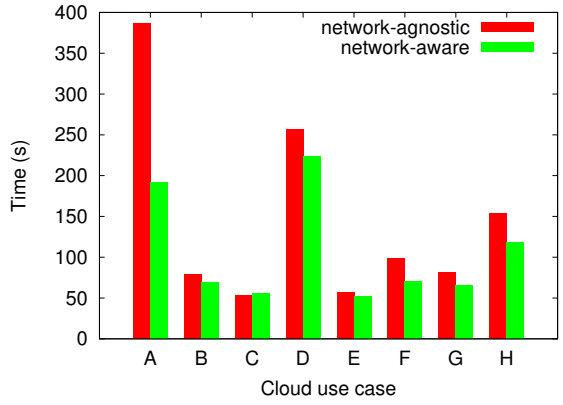


Figure 12: Network-Adapted MemEFS vs. Network-Agnostic MemEFS on 32 VMs running an I/O intensive benchmark (lower is better).

The results are consistent with the bandwidth distributions depicted in Figure 2. The use cases *A, D, F, G, H* exhibit more network bandwidth variability. The network-adapted MemEFS is therefore able to leverage this variability and improve performance. The *B, C, E* distributions indicate much more stable network conditions, thus adapting MemEFS to their bandwidth characteristics can intrinsically deliver a much smaller performance gain.

To further analyze our network adaptation policies, we selected two real-world applications, *Montage* and *BLAST*. Their parallel stages are representative for a wide range of the typical workload characteristics spectrum. First, *mProjectPP, mBackground* stages of *Montage* are CPU-bound, while *mDiffFit* is I/O bound. Next, although *BLAST* tasks are CPU-bound, they also show moderate memory and I/O utilization. Considering runtime, the *Montage* tasks are short (order of seconds), while *BLAST* tasks are longer (tens of seconds to minutes). Considering intermediary data size, while *Montage* generates small files (1-4MB), *BLAST* deals with much larger files (hundreds of MB). Therefore, we consider running these two applications sufficient to validate our network-adapted MemEFS.

Given the limited size of our cloud setup, we have scaled down the *Montage* and *BLAST* applications such that their generated data could fit in 32 VMs; their characteristics are shown in Table 2.

Figure 13a shows the *BLAST* runtimes when using the network-aware and the network-agnostic MemEFS systems in each emulated cloud use case. The results clearly show that for the more bandwidth-imbalanced setups (A, B, D, F, G, H), the network-aware MemEFS outperforms the network-agnostic MemEFS.

Figures 13b and 13c show the bandwidth in and bandwidth out utilization: the network-aware setup consistently achieves better bandwidth utilization than the network-agnostic setup. This is because in the network-agnostic setup, fast nodes are slowed down by the slow node and cannot fully utilize their bandwidth capacity. In the network-aware setup, the fast nodes hold more MemEFS partitions and thus more I/O operations are directed to them.

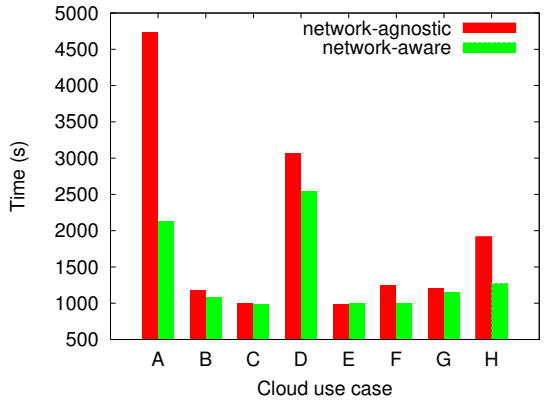
Figure 14a shows the runtimes obtained for the *Montage* workload. Figures 14b and 14c show the bandwidth utilization when running the network-aware and network-agnostic versions of MemEFS on the emulated cloud setups. The more bandwidth-imbalanced setups lead to much better performance when using the network-aware MemEFS. This is a direct consequence of the network-adapted MemEFS making better use of the available bandwidth by distributing more data to the faster nodes.

For both workflows, we notice that although the bandwidth utilization is improved in the network-aware setup, we do not reach full utilization. The explanation for this behavior is twofold. First, *BLAST* tasks are not I/O-bound. Since MemEFS I/O is done through the network, *BLAST* tasks are unable to saturate the bandwidth. Second, in the *Montage* case, the parallel stages (*mProjectPP, mDiffFit, mBackground*) are intertwined with (long-running) sequential stages (*mImgTbl, mConcatFit, mBgModel*). These synchronization points decrease the overall network utilization.

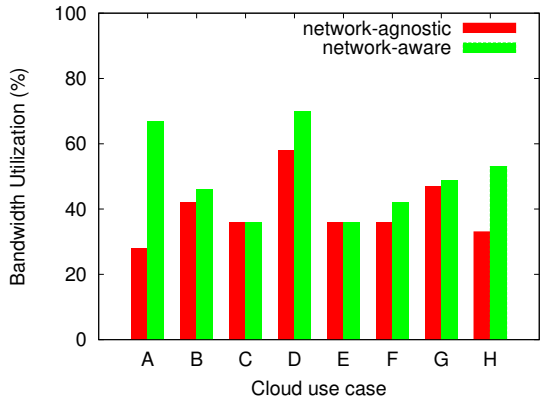
For the cloud use cases that exhibit less network variability (B, C, E), the network adaptation mechanism of MemEFS cannot achieve better bandwidth utilization since the partition-to-node mapping is similar to the network-agnostic setup. Furthermore, the network utilization achieved in this setup is close to 40% due to application characteristics: *BLAST* tasks are not I/O bound, while *Montage* has synchronization points that decrease the overall network utilization.

4.3.3. Multi-Cloud Performance Overhead

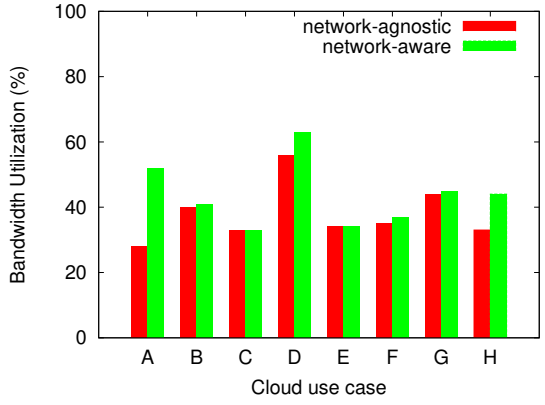
In the previous section we showed the behaviour of MemEFS when it is run on a cloud that exhibits bandwidth variability and how MemEFS adapts to such con-



(a) Total execution time (lower is better).

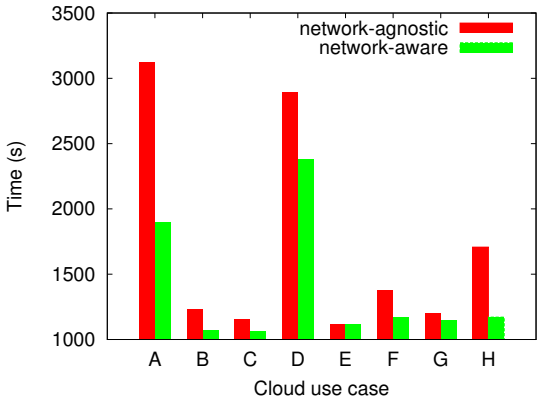


(b) Inbound Bandwidth (higher is better).

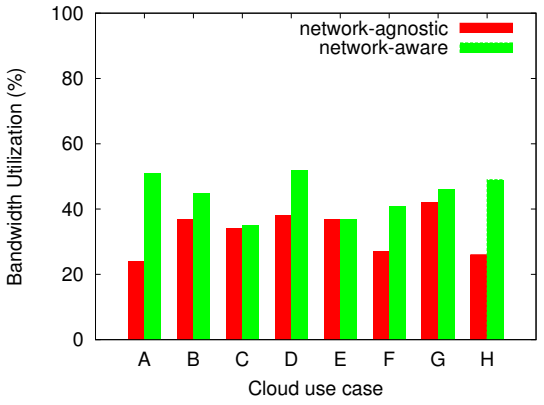


(c) Outbound Bandwidth (higher is better).

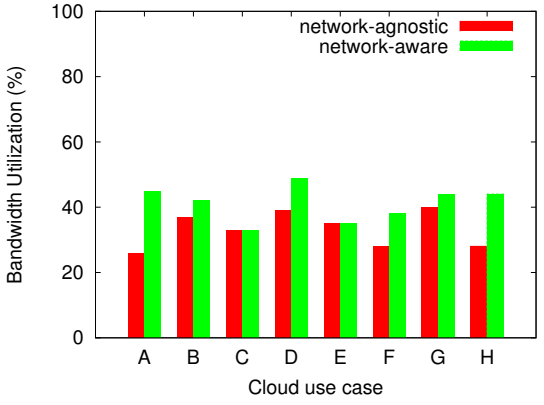
Figure 13: BLAST on 32 VMs executed using network-agnostic MemEFS and, respectively, network-adapted MemEFS.



(a) Total execution time (lower is better).



(b) Inbound Bandwidth (higher is better).

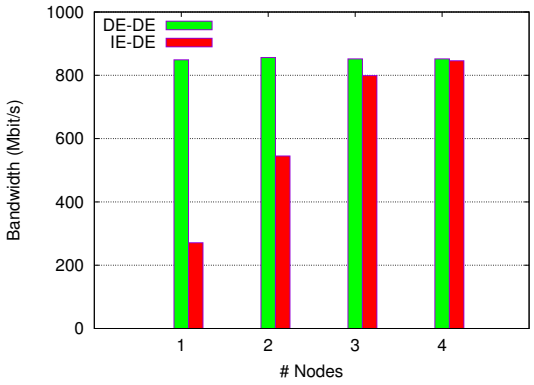


(c) Outbound Bandwidth (higher is better).

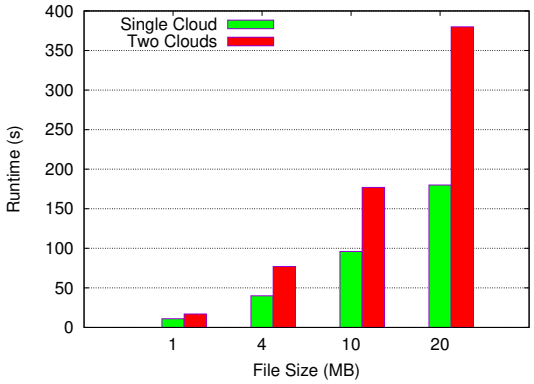
Figure 14: Montage on 32 VMs executed using network-agnostic MemEFS and, respectively, network-adapted MemEFS.

ditions. In this section, we show the behaviour of MemEFS during the migration period between a cluster (or a private cloud) and a public cloud. Such migration is usually performed in order to increase processing power and file system capacity.

To perform the experiments, we launched virtual machines in two regions of the Amazon EC2 commercial cloud: Ireland (IE) in Dublin, and Germany (DE), in



(a) Bandwidth saturation: inter-cloud (IE-DE) vs. intra-cloud (DE-DE).



(b) Application performance overhead: inter-cloud vs. intra-cloud.

Figure 15: Inter-cloud Performance Analysis

Frankfurt. Initially, we measured latency (using the *ping* tool). Intra-cloud latency is approximately 0.5 ms. Inter-cloud latency is 22.5 ms.

We measured bandwidth between the two cloud regions, as well as internal cloud bandwidth. Figure 15a plots the achieved results. The tool we used to measure bandwidth is *iperf*. The setup is as follows: we run a VM in the Germany datacenter that accepts incoming connections. Then, up to four other VMs, either from the same data center or from the remote data center, send data to the receiver VM.

We notice that when network traffic stays within the same datacenter, the incoming bandwidth measured in the receiver VM is constant at about 850 Mbit/s, irrespective of the number of senders. When the senders are located in the remote data center, only one connection between the receiver and a sender achieves about 250 Mbit/s. When the number of senders is increased, the achieved incoming bandwidth of the receiver increases as well, up to 840 Mbit/s. Therefore, we can conclude that in a many-to-many pattern of network traffic, as is the case with MemEFS, the available bandwidth can still be saturated.

We determine an upper bound on the application performance (overhead) incurred by an inter-cloud setup, by performing a network-intensive micro-benchmark. We run 4096 *dd* task on MemEFS, each individual task writing 1, 4, 10, 20 MB files. The smaller file sizes are representative for applications like Montage, while the larger file sizes are representative for applications such as BLAST. These micro-benchmarks are network-intensive workloads, while real-world applications also perform computations besides I/O. Therefore, the results obtained represent a *worst-case* scenario when running applications in an inter-cloud setup.

To run these experiments we deployed 8 *m4.xlarge* virtual machines, each having 4 virtual cores and 16 GB memory. When running on two datacenters, 4 VMs are deployed in each cloud. We compare the inter-cloud runtime with a single cloud runtime. Figure 15b plots the achieved results. The increased running times when MemEFS is deployed on two clouds are due to the large latency. For each I/O operation, the penalty of a round-trip time (45 ms) is incurred by the application runtime. As these experiments contain only I/O operations, the latency cannot be hidden and the *worst-case* scenario overhead is fully exposed.

4.4. Discussion

We have evaluated MemEFS' elastic scalability on different real-world and synthetic scientific workflows. We have designed a set of elastic scaling policies, in a range from scaling aggressively to scaling conservatively, such that the user could trade off application performance for resource utilization improvement.As expected, our results show that, with more aggressive policies both the resource usage improvement and the application slowdown decrease.

Our experiments show that **MemEFS obtains a resource utilization improvement from 47% to 65.7% on all applications, when using conservative scaling policies**. The incurred performance overhead depends on the application structure, being at most 28% for three out of four evaluated workflows. In all scenarios, **the performance overhead is much smaller than the resource utilization improvement**. The only workflow for which the performance overhead is comparable to the resource utilization improvement is BLAST, which has a highly parallel structure.

Our experiments show that users can choose from a space of different trade-offs, depending of their application structure, and performance and utilization objectives. **MemEFS' elasticity is able to reduce the re-**

source usage time (quantified as node-hours) by 8% to 22%. For users that run scientific applications repeatedly, MemEFS would greatly reduce operational costs.

When migrating scientific applications to public clouds, our experiments show that it is imperative to adapt MemEFS to the underlying network characteristics. Using real-world cloud data center bandwidth distributions [10], we evaluated MemEFS' network adaptation mechanism on two real-world applications that exhibit different structural characteristics and runtime behaviors. Our results show that the MemEFS' network adaptation mechanism greatly improves performance in cloud setups suffering from high bandwidth variability. In such scenarios, MemEFS' adaptation policy improves the application performance by up to 50% through judicious use of the available network bandwidth.

5. Related Work

We discuss three classes of related work: (i) research focused on objectives similar to ours: elasticity and in-memory data storage; (ii) research focused on some of our design issues: hashing mechanisms for better load balancing in key-value stores; and (iii) research focused on adapting workloads to the available bandwidth of the underlying network.

5.1. Elasticity in Data Storage Systems

Although elastic application scaling has gathered a lot of attention, especially with the use of IaaS clouds, research efforts were mostly focused on provisioning compute resources. However, scientific applications can process large data amounts, requiring fast access to on-demand storage. Distributed file systems, usually used to store application's data, e.g., PVFS [35], GlusterFS [36], XtreemFS [37], HDFS [38], CEPH [39], provide limited elasticity support as they are designed for cluster-wide deployments. Usually the environments in which they run are stable; node addition and removal represent the exception not the norm. These file-systems are designed with durability in mind and they employ complex data structures to optimize the storage on disks by using memory for data caching. Most of these file systems provide only manual re-balancing, requiring the intervention of an administrator. CEPH supports automatic re-balancing but with additional resource usage.

The problem of storage elasticity was addressed by Nicolae et.al [40] and Lim et.al [41]. Nicolae et.al propose an elastic storage solution for IaaS clouds in the form of a POSIX file-system. The authors share a part of our goals, mainly to minimize the wasted storage and thus the cost payed by the user while keeping the application performance overhead low. The proposed file-system provisions and releases virtual disks of fixed size from the IaaS cloud transparently to the application to meet time-varying storage demands. However, this file-system can only be used by the application running in the VM in which the file system is installed. Lim et.al. provide an elastic storage service based on HDFS that provisions nodes from a cloud provider and uses them for storage capacity. The authors use the CPU utilization of the storage nodes as a metric to change the number of provisioned nodes, considering that this metric is correlated to the performance of the storage service, e.g., response time per request. When the number of storage nodes is changed, data re-balancing is also performed, with the goal of optimizing CPU utilization and I/O bandwidth. This solution adapts the number of storage nodes to improve application data access time, while MemEFS adapts its number of nodes to total application storage demand.

Faster data access can be achieved by distributing the data across the memory of the nodes on which the application is running. RamCloud [42] and FaRM [43] provide different optimized means for applications to store their data in memory. Other distributed in-memory caching systems, based on memcached, were proposed as an intermediate layer between the applications and the distributed file systems, to speed up the access to data [44, 45]. However, these solutions were designed to be deployed on the entire cluster and they lack elasticity support. AMFS [4] or MemFS [6] provide generic in-memory runtime file systems but are also designed to run on a static number of nodes.

Elasticache [46] and Hazelcast [47] provide elastic in-memory caching services based on memcached and Redis. Although they allows users to add more nodes to the in-memory cache cluster, they lack automated load-balancing and auto-scaling mechanisms to change the number of nodes based on dynamic application storage demand.

5.2. Load Balancing Schemes in DHTs

Several works focused on load balancing techniques for key-value stores [48, 49, 50, 22]. The most promising class of solutions is based on consistent hashing.

ZHT is a zero-hop distributed hash table [48]. As in MemEFS, ZHT keys are assigned to partitions which are then distributed over physical nodes. Each node has one or more ZHT instances, each of them maintaining one or more partitions and serving requests for

them. ZHT supports heterogeneous systems with various storage capacities and computing power by varying the number of partitions per node. However, ZHT provides poor elasticity support in contrast to MemEFS, for which node addition or removal is fully automated. When a new node joins ZHT, it adds itself in the ring as the neighbor of the most loaded node and starts migrating partitions from its neighbor. Node departures are done manually: for a node to leave the system, an administrator needs to modify the global membership table. Furthermore, a detailed comparison between MemEFS and ZHT is outside the scope of this paper, as it would translate to comparing ZHT and Redis. This is because MemEFS offers a POSIX-like interface to its data-store (Redis), while ZHT only offers a simple key-value interface.

Other works rely on the use of two hash functions [49, 50, 22]: one to map the nodes to a continuous interval $[0, 1)$ and another one determine the location of the keys by mapping them in the same interval. Brinkmann et al. introduces two adaptive hashing strategies [49] to redistribute keys among nodes when the capacities of the nodes, the number of nodes or the number of keys change. Each node is in charge of multiple virtual bins, each virtual bin handling one subinterval with a length proportional to its capacity and a stretch factor. Schindelhauer et al. improves the load balancing in heterogeneous DHT by choosing nodes for keys based on weights [50]. Each node is assigned a positive weight and keys are distributed to it with a probability proportional with the node's weight and inversely proportional with the sum of all node weights. To provide elasticity and cope with node heterogeneity, MemEFS adapts Y0's algorithm [22]. Opposed to these previous solutions, Y0 gives MemEFS more flexibility in deciding the total number of partitions, allowing a more fine-grain control on how much MemEFS should scale.

5.3. Network Adaptability

We identified several studies that improve application performance when the underlying compute resources suffer from bandwidth variability. The main difference between our approach and these studies is that we target the distributed file system level. Therefore, our approach is more generic and transparent to the application and scheduler.

In [51], the authors propose the use of a *Software Defined Network* (SDN) to achieve a bandwidth-aware scheduler for Hadoop. They utilize the link measuring and bandwidth setting capabilities of an SDN to distribute data and tasks in such a way that the makespan of a MapReduce job is minimized. Another framework [52] that adds network awareness to Hadoop considers multi-cluster Hadoop setups. Because inter-cluster bandwidth is often lower than intra-cluster bandwidth, they account for this in their scheduler and achieve good performance in terms of makespan. *EHadoop* is another framework that also takes into account the network usage of MapReduce jobs when scheduling [53]. It decouples data from computation by having two separate clusters. The bandwidth between the two clusters is variable. By performing online profiling of task network usage and completion time, EHadoop keeps job completion time stable when faced with different network topologies.

In [54], the authors describe a task scheduler for independent tasks that incorporates bandwidth knowledge to schedule tasks on resources. Tasks are scheduled on VMs with different bandwidth capabilities. It is not immediately clear, however, if the bandwidth requirements of the individual tasks are known beforehand. Assuming available bandwidth is known, but variable, the authors from [55] propose a DAG workflow scheduler that minimizes makespan using fuzzy optimization techniques. It can handle workflows that have inter-task data dependencies, such as Montage. It assumes data transfer information between tasks is known beforehand.

6. Conclusions

Scientific workflows exhibit significant storage demand variability at runtime. To overcome this issue, traditional approaches generally over-provision the number of storage nodes, such that the system could handle the peak storage demand. Our contribution, MemEFS, scales elastically at runtime, transparently to the application and based on the storage demand, while distributing data across system nodes to achieve load balance for both storage and network traffic.

With the rapid increase of data volumes generated by scientific domains such as bioinformatics or astronomy ("data deluge"), we expect scientific workflows to outgrow the (memory) storage capacities of private clusters. As a consequence, clusters will need to be augmented by means of public cloud computing infrastructure. However, as many studies point out, such platforms are plagued by large bandwidth variability due to colocation and virtualization overheads.

Our experiments show that it is imperative to adapt the storage layer to the underlying network, as otherwise the application would observe a severe performance penalty. To overcome this performance penalty,

we equipped MemEFS with a mechanism that leverages network variability and increases performance by up to 50%. Our experiments show that the larger the network variability is, the more MemEFS outperforms the network-agnostic approach.

We show that, with simple adaptation policies, MemEFS replaces the need to either a-priori estimate application resource requirements or to over-provision resources. Our experiments show that MemEFS able to largely improve resource utilization, while incurring only modest performance overheads. Our results are a promising step in further exploring trade-offs between resource utilization and application performance.

Acknowledgments

This work is partially funded by the Dutch public-private research community COMMIT/. The authors would like to thank Kees Verstoep for providing excellent support on the DAS-4 clusters.

References

[1] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. Prince, et al., Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking, International Journal of Computational Science and Engineering 4 (2) (2009) 73–87.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, D. J. Lipman, Basic local alignment search tool, Journal of molecular biology 215 (3) (1990) 403–410.

[3] W. Gropp, E. Lusk, A. Skjellum, Using MPI: portable parallel programming with the message-passing interface, Vol. 1, MIT press, 1999.

[4] Z. Zhang, D. S. Katz, T. G. Armstrong, J. M. Wozniak, I. Foster, Parallelizing the Execution of Sequential Scripts, in: High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for, IEEE, 2013.

[5] A. Uta, A. Sandu, T. Kielmann, Overcoming data locality: An in-memory runtime file system with symmetrical data distribution, Future Generation Computer Systems 54 (2016) 144–158.

[6] A. Uta, A. Sandu, T. Kielmann, MemFS: an In-Memory Runtime File System with Symmetrical Data Distribution, in: IEEE Cluster, 2014, pp. 272–273, (poster paper).

[7] A. Uta, A. Sandu, S. Costache, T. Kielmann, Scalable in-memory computing, in: Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on, IEEE, 2015, pp. 805–810.

[8] A. J. Hey, S. Tansley, K. M. Tolle, et al., The fourth paradigm: data-intensive scientific discovery, Vol. 1, Microsoft research Redmond, WA, 2009.

[9] J. Schad, J. Dittrich, J.-A. Quiané-Ruiz, Runtime measurements in the cloud: observing, analyzing, and reducing variance, Proceedings of the VLDB Endowment 3 (1-2) (2010) 460–471.

[10] H. Ballani, P. Costa, T. Karagiannis, A. Rowstron, Towards predictable datacenter networks, in: ACM SIGCOMM Computer Communication Review, Vol. 41, ACM, 2011, pp. 242–253.

[11] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, R. Chaiken, The nature of data center traffic: measurements & analysis, in: Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference, ACM, 2009, pp. 202–208.

[12] T. Benson, A. Akella, D. A. Maltz, Network traffic characteristics of data centers in the wild, in: Proceedings of the 10th ACM SIGCOMM conference on Internet measurement, ACM, 2010, pp. 267–280.

[13] A. Uta, A. Sandu, S. Costache, T. Kielmann, MemEFS: an elastic in-memory runtime file system for escience applications, in: e-Science (e-Science), 2015 IEEE 11th International Conference on, IEEE, 2015, pp. 465–474.

[14] M. Szeredi, et al., FUSE: Filesystem in userspace, `http://fuse.sourceforge.net/`.

[15] B. Aker, Libmemcached, `http://libmemcached.org\/libMemcached.html` (2016).

[16] B. Fitzpatrick, Distributed Caching with Memcached, Linux journal 2004 (124) (2004) 5.

[17] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. Maechling, R. Mayani, W. Chen, R. da Silva, M. Livny, K. Wenger, Pegasus: a workflow management system for science automation, in: Journal of Future Generation Computer Systems, Elsevier, 2015.

[18] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, D. Lewin, Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web, in: Twenty-ninth annual ACM symposium on Theory of computing, ACM, 1997, pp. 654–663.

[19] xxhash, `https://code.google.com/p/xxhash/` (2016).

[20] D. Eastlake, P. Jones, Us secure hash algorithm 1 (sha1) (2001).

[21] R. Rivest, The md5 message-digest algorithm (1992).

[22] P. B. Godfrey, I. Stoica, Heterogeneity and load balance in distributed hash tables, in: INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies., Vol. 1, IEEE, 2005, pp. 596–606.

[23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet applications, in: ACM SIGCOMM Computer Communication Review, Vol. 31, ACM, 2001, pp. 149–160.

[24] Apache Libcloud, `https://libcloud.apache.org` (2016).

[25] S. Sanfilippo, P. Noordhuis, Redis, `http://redis.io` (2014).

[26] hiredis, `https://github.com/redis/hiredis` (2016).

[27] P. Hunt, M. Konar, F. P. Junqueira, B. Reed, ZooKeeper: wait-free coordination for internet-scale systems, in: USENIX annual technical conference, Vol. 8, 2010, pp. 11–11.

[28] SCEC project, Southern California Earthquake Center, `http://www.scec.org/` (2015).

[29] Pegasus Workflow Generator, `https://confluence.pegasus.isi.edu/display\/pegasus/WorkflowGenerator` (2016).

[30] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, K. Vahi, Characterizing and profiling scientific workflows, Future Generation Computer Systems 29 (3) (2013) 682–692.

[31] Z. Zhang, D. Katz, Using application skeletons to improve escience infrastructure, in: e-Science (e-Science), 2014 IEEE 10th International Conference on, Vol. 1, 2014, pp. 111–118.

[32] DAS-4, The Distributed ASCI Supercomputer, `http://www.cs.vu.nl/das4/` (2016).

[33] Open Nebula, `http://www.opennebula.org` (2016).

[34] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, Y. Zhang, Secondnet: A data center network virtualization architecture with bandwidth guarantees, in: Proceedings of the 6th International COnference, Co-NEXT '10, 2010, pp. 15:1–15:12.

[35] R. B. Ross, R. Thakur, et al., PVFS: A parallel file system for

linux clusters, in: 4th Annual Linux Showcase and Conference, 2000, pp. 391–430.
[36] GlusterFS, http://www.gluster.org/ (2016).
[37] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, E. Cesario, The XtreemFS Architecture - A Case for Object-based File Systems in Grids, Concurrency and computation: Practice and experience.
[38] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop Distributed File System, in: Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, IEEE, 2010, pp. 1–10.
[39] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, C. Maltzahn, Ceph: A scalable, high-performance distributed file system, in: 7th symposium on Operating systems design and implementation, USENIX Association, 2006, pp. 307–320.
[40] B. Nicolae, P. Riteau, K. Keahey, Bursting the Cloud Data Bubble: Towards Transparent Storage Elasticity in IaaS Clouds, in: IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS '14, 2014, pp. 135–144.
[41] H. C. Lim, S. Babu, J. S. Chase, Automated control for elastic storage, in: 7th International Conference on Autonomic Computing, ICAC '10, 2010, pp. 1–10.
[42] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, et al., The case for RAMCloud, Communications of the ACM 54 (7) (2011) 121–130.
[43] A. Dragojevic, D. Narayanan, M. Castro, O. Hodson, FaRM: Fast Remote Memory, in: 11th USENIX Symposium on Networked Systems Design and Implementation, 2014, pp. 401–414.
[44] N. S. Islam, X. Lu, M. Wasi-ur Rahman, R. Rajachandrasekar, D. K. Panda, In-memory i/o and replication for hdfs with memcached: Early experiences, in: 2014 IEEE International Conference on Big Data, IEEE, 2014, pp. 213–218.
[45] F. R. Duro, J. G. Blas, J. Carretero, A hierarchical parallel storage system based on distributed memory for large scale systems, in: 20th European MPI Users' Group Meeting, EuroMPI '13, ACM, New York, NY, USA, 2013.
[46] Amazon ElastiCache, http://aws.amazon.com/elasticache/ (2016).
[47] Hazelcast, http://http://hazelcast.com/ (2016).
[48] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, I. Raicu, ZHT: A light-weight reliable persistent dynamic zero-hop distributed hash table, in: Parallel & Distributed Processing Symposium (IPDPS), 2013.
[49] A. Brinkmann, K. Salzwedel, C. Scheideler, Compact, Adaptive Placement Schemes for Non-uniform Requirements, in: 14th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '02, ACM, New York, NY, USA, 2002, pp. 53–62.
[50] C. Schindelhauer, G. Schomaker, Weighted Distributed Hash Tables, in: ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '05, ACM, New York, NY, USA, 2005, pp. 218–227.
[51] P. Qin, B. Dai, B. Huang, G. Xu, Bandwidth-aware scheduling with sdn in hadoop: A new trend for big data, arXiv preprint arXiv:1403.2800.
[52] P. Kondikoppa, C.-H. Chiu, C. Cui, L. Xue, S.-J. Park, Network-aware scheduling of mapreduce framework ondistributed clusters over high speed networks, in: Proceedings of the 2012 workshop on Cloud services, federation, and the 8th open cirrus summit, ACM, 2012, pp. 39–44.
[53] L. Yazdanov, M. Gorbunov, C. Fetzer, Ehadoop: network i/o aware scheduler for elastic mapreduce cluster, in: Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on, IEEE, 2015, pp. 821–828.
[54] W. Lin, C. Liang, J. Z. Wang, R. Buyya, Bandwidth-aware divisible task scheduling for cloud computing, Software: Practice and Experience 44 (2) (2014) 163–174.
[55] C. G. Chaves, D. M. Batista, N. L. da Fonseca, Scheduling cloud applications under uncertain available bandwidth, in: Communications (ICC), 2013 IEEE International Conference on, IEEE, 2013, pp. 3781–3786.