MemEFS: an Elastic In-Memory Runtime File System for eScience Applications

Alexandru Uta, Andreea Sandu, Stefania Costache, Thilo Kielmann
Dept. of Computer Science, VU University Amsterdam, The Netherlands
a.uta@vu.nl, a.sandu@vu.nl, s.v.costache@vu.nl, thilo.kielmann@vu.nl

***Abstract*—Data-intensive scientific workflows exhibit inter-task dependencies that generate file-based communication schemes. In such scenarios, traditional disk-based storage systems often limit overall application performance and scalability. To overcome the storage bottleneck, in-memory runtime distributed file systems speed up application I/O. Such systems are deployed statically onto a fixed number of compute nodes and act as a distributed, fast I/O cache for the runtime generated data. Such static deployment schemes have two major drawbacks. First, the user is faced with the sometimes difficult task of estimating the size of the generated data, as the application would fail otherwise. Second, because applications exhibit significant variability of the data footprint and of the achieved parallelism during their runtime, this deployment scheme also leads to severe resource under-utilization. To address these limitations, we present MemEFS, an elastic in-memory runtime distributed file system. MemEFS is able to scale elastically, based on application storage demands, by acquiring or releasing resources when needed. Our evaluation shows that, while generating modest runtime overheads, MemEFS is able to increase the resource utilization efficiency by up to 65%.**

I. Introduction

An important direction of the current eScience research focuses on *efficiently* running *scientific workflows*. These computations are usually composed of many data-intensive tasks and span over multiple domains, ranging from data analytics to domain-specific scientific computations, e.g., astronomy [1] or bioinformatics [2]. Such applications exhibit inter-task dependencies expressed by means of files (i.e. the output of one task is the input of another). Opposed to traditional message passing mechanisms [3], this communication scheme requires a *shared*, or *distributed, file system*. However, *data-intensive* scientific workflows generate large data amounts, which cannot be efficiently handled by the traditionally used disk-based distributed file systems, thus leading to a limited application performance and scalability.

To alleviate the storage bottleneck, state-of-the-art [4], [5] suggests using in-memory runtime distributed file systems. Currently, 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. Even though performance is an important aspect when running *scientific workflows*, the *applicability* and *efficiency* of in-memory runtime distributed file systems remains *limited* as they are *statically deployed* onto a fixed number of compute nodes.

We believe that an *elastic* scheme that allocates or deallocates resources based on application demands offers more *flexibility* and a larger *optimization space* to the users of such systems. A direct implication 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 data at runtime. In the case of a static deployment, the user would be faced with the sometimes difficult task of estimating the storage demands of the application before starting the application. An under-estimation would lead to poor performance (i.e. 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 of the data footprint and of the achieved parallelism. The former is determined by two aspects: (i) data-intensive parallel stages generate large amounts of data; (ii) runtime generated data can be discarded when it is no longer needed. The latter is a result of the mix of large parallel stages with sequential synchronization points, e.g., data aggregation, data partitioning stages. Under such conditions, a static deployment scheme needs to overprovision resources to accommodate the peak storage demands of the application. This leads to *poor resource utilization* as nodes cannot be added or removed on-demand, during runtime. In shared clusters, such behavior translates to longer queueing times for users and inefficient energy consumption, assuming machines that are no longer used could be powered off.

In this work, we introduce **MemEFS**, a locality-agnostic in-memory elastic storage system that builds on our previous research [6], [5]. In contrast to static deployment schemes, MemEFS is able to scale dynamically, at runtime, based on the application storage demands. Thus, MemEFS significantly reduces resource inefficiencies generated by overprovisioning for peak storage demands. The contributions of this paper are two-fold:

- We propose the design of MemEFS, an elastic in-memory storage system. MemEFS distributes the data uniformly across nodes through the use of a two-layer hashing scheme, while storing data in a key-value store [7]. MemEFS scales elastically during runtime while maintaining a good load balance for both storage and network traffic.
- 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 based on application storage demand. Our results show that MemEFS improves resource utilization (by up to 65% in most situations) with modest performance impact. Furthermore, our policies allow the user to trade off resource efficiency for performance.

This paper is organized as follows. Section 2 presents the background of our work, while Section 3 introduces the design of MemEFS. Section 4 describes the evaluation results and Section 5 discusses related work. Finally, Section 6 concludes the paper.

II. Background

In our previous work [6], [5] 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 attractive 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) and it maximizes the achieved bandwidth and throughput for reading and writing operations. MemFS balances its load among the nodes through the use of a key-value store. In MemFS, 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 allows it to improve the 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, supercomputers, or clouds. Hence, all-to-all connectivity is ensured and, before runtime, 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 the case of Distributed Hash Tables (DHTs), which are designed for environments where nodes join and leave the system at very high rates, such a small complexity is not possible. As in such environments it would be too expensive for every node to keep a full view of the network, only a partial view is used. Thus, nodes use routing tables to be able to direct queries, leading to a complexity of $O(\log n)$ for look-up operations.

MemFS's design is based on three components: (1) a FUSE [8] layer that serves as a POSIX interface to applications and which handles the file striping mechanism; (2) the Libmemcached [9] hashing protocol which determines in $O(1)$ steps which node holds a certain file stripe; and (3) Memcached [10], a fast, in-memory key-value data store.

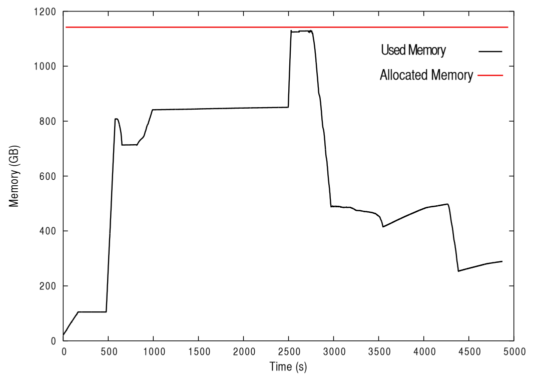


Fig. 1. The in-memory storage demand of a scientific workflow during its runtime. Note the difference between the used storage and the allocated amount to satisfy the peak demand.

A. Elasticity Requirements

Current in-memory file systems, like MemFS, lack support for scaling their number of nodes to application storage needs. Nevertheless, scientific workflows often have varying storage demands during their runtime. Let us take for example an astronomy workflow, Montage [1], which builds 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 of the data from the previous stage, a part of the runtime data can be deleted by the workflow manager [11], to optimize the total data storage used by the workflow.

Figure 1 describes the variations in used 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 a user would have to run Montage on a static number of nodes, she would have to provision for the peak memory utilization (depicted in Figure with red). Not only she will have to know this maximum utilization before starting Montage, but also, due to varying parallelism levels of 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 partitioning and load balancing mechanisms. Load balancing after a reconfiguration implies moving data among 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. To maintain the storage balancing properties, the keys would need to be rehashed and data would need to be migrated among nodes. However, the movement of data during load balancing 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 ideas behind 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 among nodes when nodes are added or removed; (ii) and elastic scaling policies to adapt the number of nodes to variations in application storage demand.

III. MemEFS Design and Implementation

Figure 2 gives an overview of MemEFS. Our elastic in-memory distributed file system consists of worker nodes and a Central Manager that gathers worker node statistics for taking reconfiguration decisions 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 Central Manager. The Central Manager could run on a separate node, but it can also reside on a worker node without affecting the worker's performance.

As previously explained, to achieve load balancing, when new nodes are added, MemEFS needs to move data. To minimize the data movement, MemEFS uses a consistent hashing scheme [12]. 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 believe 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. With this argument in mind, each node holds multiple partitions, such that, when reconfiguring the file system, we migrate partitions, thus avoiding 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. The size of each partition is limited by the total amount of memory the nodes have. 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.

A. 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 [13] 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, leveraging up to 13GB/s throughput [13], outperforming other hashing algorithms like SHA1 [14] or MD5 [15] 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.

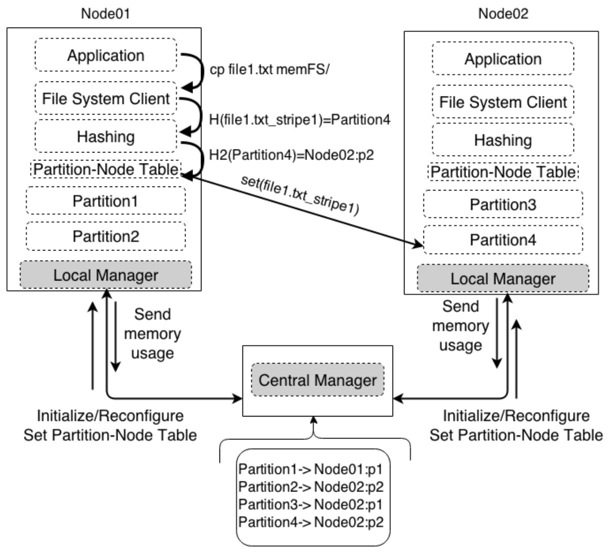


Fig. 2. Architecture of MemEFS.

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 Central manager at each reconfiguration. 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.

B. Load Balancing

To compute the number of partitions each node stores after each reconfiguration, MemEFS adapts an algorithm proposed in Y0 [16]. Y0 improves the Chord [17] DHT to achieve good load balance, even with nodes that are heterogeneous in terms of storage. The authors have shown that the load imbalance in Y0 is 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 equal to:

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

where c_v is the node's normalized capacity such that $\sum_{v \in Nodes} c_v = n$, and f_v is the fraction of the storage space assigned to node v. A system is thus load balanced when each node's share is equal to 1. Thus, the value of f_v gives the system's load balance. The authors show that, when each node holds $2\log(n)$ partitions per capacity unit, the system load balance is at most 3.6.

To achieve load balance in MemEFS, we adapt the Y0's computation of nodes' shares to our needs. We define the capacity of each node v, as:

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

Then, the share of node v becomes:

$$share(v) = \frac{P_v/P}{c_v/n} \tag{3}$$

where P_v is the number of partitions of node v and P is the total number of partitions from the system, $P = \sum_{v \in Nodes} P_v$.

If MemEFS would use only $2 \log n$ partitions per node when scaling out, it would be limited to adding at most $2n \log n$ nodes to the system. Moreover, in [16], the authors show that for higher number of partitions, the imbalance factor is even lower than 3.6. However, in the case of Y0, larger number of partitions per capacity unit add significant overhead for storing the finger table and look-up operations. MemEFS is not affected by this, as it achieves $O(1)$ look-up operations. Thus, when defining the number of partitions of a node, P_v, instead of using $2 \log n$ partitions per capacity unit, we define a *scaling* constant β, such that:

$$P_v = c_v \beta \log n \tag{4}$$

This constant gives users more control in defining how much MemEFS should scale out.

C. Initialization and Reconfiguration

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

1) ***File System Initialization****:* When a user starts MemEFS, a node running the *Central Manager* has to be started. The manager process takes as input the number of initial worker nodes and β (for all the experiments in this paper we have used $\beta = 4$). Then, the manager starts the worker nodes. Based on the load balancing scheme introduced in Section III-B, the manager creates the required numbers of partitions on the worker nodes. 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.

2) ***File System Reconfiguration****:* When an application is running, the 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. After the new workers have been started, or before existing workers need to be removed, the Central Manager 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 Central Manager removes a part of the current nodes if the memory utilization for a certain period of time, e.g., 45 seconds, does not increase, and is below a given threshold. We give more details about the policies used to add and remove nodes in Section IV.

Because during reconfiguration 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.

D. Implementation

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

1) ***Data Store****:* To store file stripes, MemEFS relies on existing key-value data stores. However, an important mechanism that MemEFS requires is to allow it to partition the data and migrate partitions among nodes. In our previous work, we have 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, for MemEFS we have opted to use Redis as a key-value store [7]; 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) 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; (iii) master-slave replication; we can achieve 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.

MemEFS uses the first mechanism as it was the most advantageous. The second mechanism has two disadvantages: usually log files are larger than the database dumps, and replaying the log file might be more time consuming than loading the dump file. Using the last mechanism is difficult because there is no exact mechanism to check when the replication has finished. Moreover, during the replication process, the initial Redis process also generates a database dump, similar to the first mechanism.

2) ***Communication between Central Manager and Workers****:* Each *worker node* runs a *Local Manager* process that communicates with the *Central Manager*. Figure 3 depicts the communication protocol. The *Local Manager* process runs three threads. The first thread measures the local memory utilization at regular time intervals; the time interval is configurable with a default value of one second. The second thread sends the local memory utilization information to the *Central Manager* whenever this information is requested. The 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 File System Client acknowledges that the application's I/O operations have been suspended, the Local Manager sends an acknowledgment back to the *Central Manager* signaling that the reconfiguration can take place. When the reconfiguration has finished, the *Central Manager* sends a message with the new Partition-to-Node mapping to the Local Manager. When the Local Manager receives this message, it sends another signal to the File System Client, letting it know that it should now reload its Partition-to-Node table and then safely resume the application's I/O operations.

*3) **File System Client**:* The MemEFS file system client is implemented as a FUSE module which communicates with Redis using the *hiredis* communication library [18]. We have leveraged the implementation of MemFS's file system client and adapted it to support elasticity and suspend/resume for the application's I/O operations during reconfigurations. When receiving the reconfiguration signal, the File System Client waits until the current *read()*, *write()* or other application requests finish, then blocks all other incoming application requests, i.e., implemented in the request handling code as a wait on a semaphore, and sends back an acknowledgment. When the File System Client receives the signal that informs it to resume the application operations, it first recomputes the Partition-to-Node table and then resumes the application's I/O operations, i.e., by incrementing the sempahore.

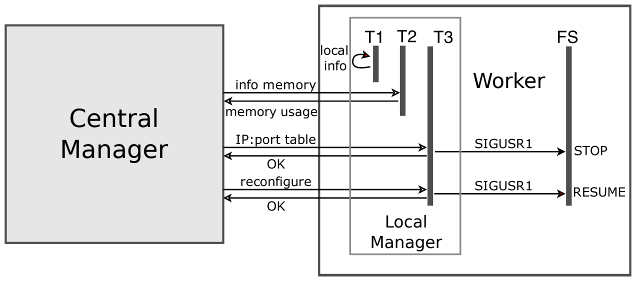


Fig. 3. Communication between the Central Manager and a worker.

*4) **Fault-Tolerance**:* MemEFS can be configured to be 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. Nevertheless, Redis 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 and also slows down the writing operations. The Central Manager of MemEFS can be made highly available by leveraging state-of-the-art solutions [19]. We leave for future work the different strategies for fault-tolerance, such as erasure coding for data and state machine replication for the Central Manager.

IV. Evaluation

In this section we present the evaluation of MemEFS. We show how MemEFS improves the resource utilization efficiency and how its elastic scaling affects the user perceived performance when running scientific workflows. We evaluate a set of elastic policies, which prove that, for various workflows, MemEFS can provide different trade-offs between resource utilization efficiency and application performance.

A. Experimental Setup

The experiments were executed on our local DAS4 multi-cluster system [20]. The (in total 74) compute nodes are equipped with dual-quad-core Intel E5620 2.4 GHz CPUs and 24GB memory. The nodes are connected using 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 achieves 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 and the applications. 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 our 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. 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 show how real-world applications can benefit from it.

In our experiments we used two real-world and two synthetic scientific workflows. Table I describes the characteristics of these workflows in terms of total number of tasks, size of input data and maximum amount of occupied storage during their runtime. The two real-world workflows we used are Montage [1] and BLAST [2], as 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 we evaluated are Broadband [21] and Cybershake [21]. Because the access to their code or used input data is not open, we generated the two synthetic workflows using execution traces of their real-world counterparts taken from [22] and [23]. From the workflows described in [22], [23] these two generated the largest data amounts. To generate the synthetic workflows, we used the

TABLE I
WORKFLOW CHARACTERISTICS

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

Application Skeletons framework [24]. This framework allows the user to specify the data usage patterns and runtimes of application tasks, together with dependencies between them.

B. Elastic Scaling Policies

To show how MemEFS can scale elastically based on application storage demands, we designed several *elastic scaling* policies. The policies can scale out or in the number of storage (and also compute) 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. These policies offer trade-offs between saving more resources and workflow execution speed.

For scaling out we define three policies, which range from a conservative to an aggressive scale out. If the *scaling out* policy is more conservative, i.e., scales with small number of nodes, the system also benefits from less compute resources as in MemEFS, the storage is co-located with the compute nodes. Hence, the more conservative the *scaling out* policy is, the higher is 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: scale in conservatively or aggressively. The more aggresive the *scaling in* policy is, the higher is 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.

For applications that only exhibit an increase in data usage we only evaluate the scaling out policies. For applications for which we delete runtime generated data that is no longer needed, we combine the scaling out with the scaling in policies.

To illustrate the behavior of the policies, Figure 4 depicts the resource utilization and allocated amount during the runtime of Montage for all the six possible policy combinations. Due to space constraints, we do not present the elastic scaling behavior of the other evaluated applications. However, we present in the next section their performability evaluation to show how the scaling policies affect the resource utilization efficiency and the application performance.

C. MemEFS Performability

We ran the workflows on MemEFS using two deployment schemes: static and elastic. In the *static* deployment scheme, MemEFS provisions enough nodes to store all data generated by the application. In the *elastic* deployment scheme initially MemEFS provisions enough nodes to copy the application input data and afterwards it uses the previously discussed scaling policies. The results presented in the remainder of this section are obtained from averaging 4 experiment runs.

To evaluate the elastic policies we use the following performability metrics:

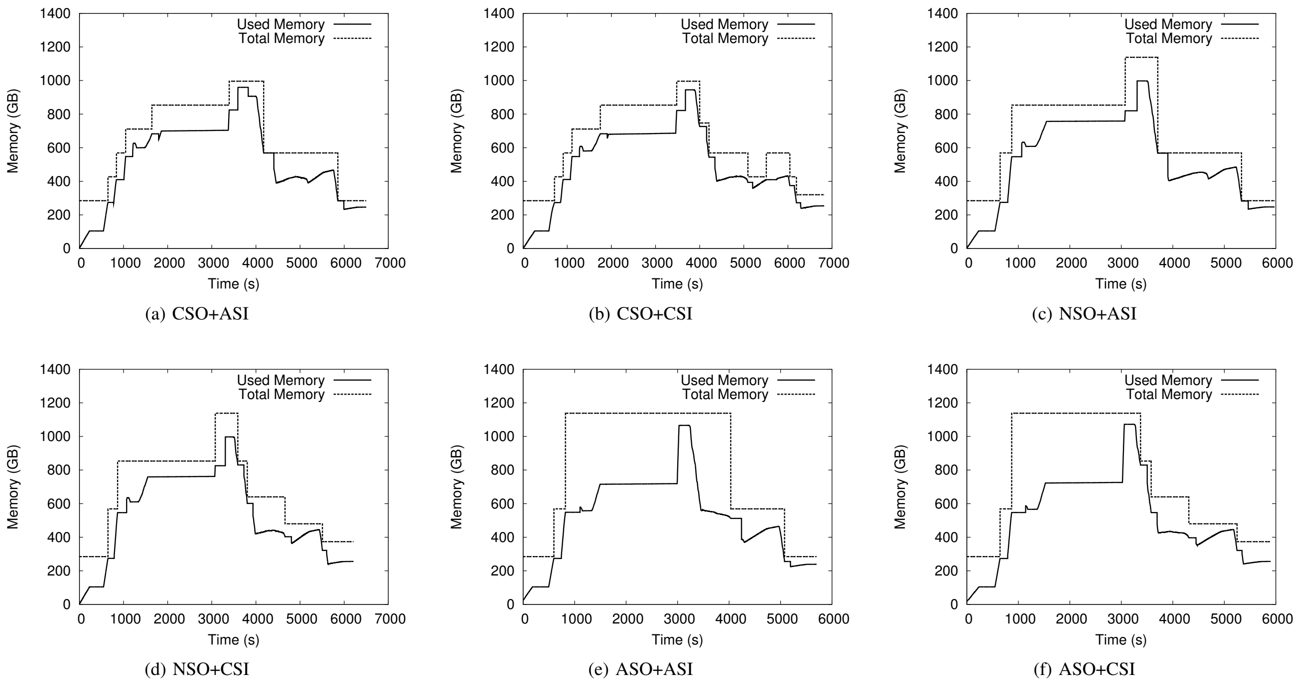
- **Resource Usage Improvement:** represents the amount of resources (memory, nodes) wasted for an elastic run compared to the amount of resources wasted for the static run. The resource usage improvement is defined by:

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

  where $W(s)$ represents the amount of wasted resources in a deployment scheme s. We define this amount as $W(s) = A(s) - U(s)$, where $A(s)$ is the amount of allocated resources, and $U(s)$ is the amount of used resources.
- **Performance Overhead:** represents how much slower is an elastic run compared to the static run. This overhead is given by elastically running MemEFS, since an elastic run adds reconfiguration overheads and also leads to less compute capacity for the application.

Figure 5a presents the results of our elastic policies for a run of Montage. We sorted the plot bars based on the policy aggressivity. During the runtime of Montage we delete the intermediate data that is no longer needed 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 aggresive policies use more workers during runtime, thus achieving a better application speedup. For Montage, the resource usage improvement varies between 31.9% (**ASO+ASI**) and 65.7% (**CSO+CSI**), while the performance overhead varies between 14.4% (**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 while not adding performance overhead compared to the static Montage run.

Figure 5b shows the policies evaluation when running BLAST. For BLAST, we cannot delete any data generated



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

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

Fig. 4. The behavior of elastic scaling policies for Montage.

during runtime, thus we evaluate only the scaling out policies. As expected, when the policy aggressivity increases, both the assessed metrics show a decreasing trend, with the performance overhead being proportional to the resource utilization improvement. In this case, the resource utilization improvement 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. Thus, running with less worker nodes also slows down the execution.

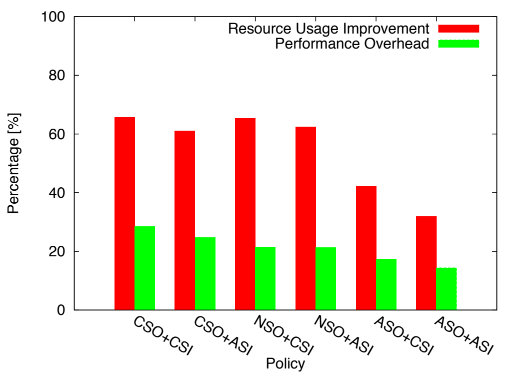
Figure 5c presents the policies evaluation for the synthetic Broadband workflow. Similarly to BLAST, we could not delete data generated during runtime, thus we could not evaluate the scaling in policies. As for the previous applications, we also notice a decreasing trend in both metrics while the policy aggresivity 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**). For Broadband, it is interesting to notice that the **CSO** policy is a clear winner with a 55.6% resource usage improvement while showing only a 22.2% performance overhead. This is explained by the Broadband workflow structure as presented in [23]. In the first stages, the workflow is less parallel, while the final stages exhibit large parallelism. Thus, the CSO policy largely decreases the resource utilization for the first stages while not slowing down too much the application.

Figure 5d shows the policies evaluation for the synthetic Cybershake workflow. Because Cybershake needed a larger storage for its input, we started the 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 behave similarly. Thus, for these policies we notice the same values for both resource utilization improvement and performance overhead metrics. Since for Cybershake we were able to delete data that is no longer needed during runtime, we also combined the scaling out policies with the scaling in policies. As expected, when the aggresivity of the policy increases, the evaluated metrics show a decreasing trend in resource utilization improvement and performance overhead. Like in the case of Broadband, we notice that the conservative policy is a clear winner. The **CSO+CSI** saves 47.04% resources while it decreases the runtime by only 19.9%. The explanation for this behavior is similar to the Broadband case: Cybershake exhibits less achievable parallelism in the first workflow stages. Hence, the CSO policy is able to save more resources while not decreasing too much the runtime.

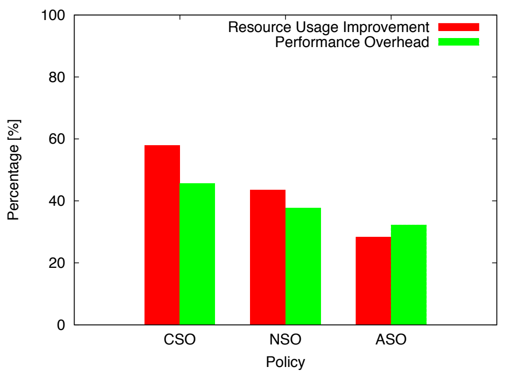
D. Storage Load Balancing

In Figure 6 we present our analysis of MemEFS storage load balance during an elastic run. We selected Montage for this analysis, which we ran with a smaller input than in the previous experiments. The experiment starts on 8 nodes and the highest number of used nodes is 24. The experiment finishes on 12 nodes. The elastic scaling policy used in this experiment is **NSO+ASI**. Hence, the system scales out twice and scales in only once. We measured the memory load of all the nodes in the system at each second. To measure the load imbalance, we computed the average and standard deviation.

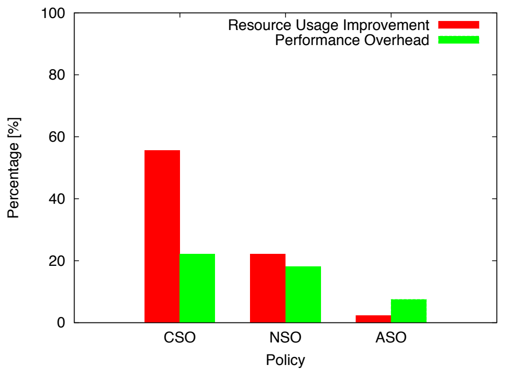
Figure 6 shows the average node memory utilization and the standard deviation for each 1 second time interval. We notice that the difference between the average and standard deviation is at most 17%. This imbalance of 17% is given by the partition granularity. In this case, the initial number



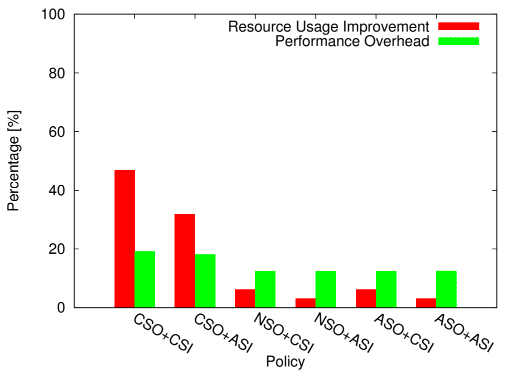
(a) Montage.



(b) BLAST.



(c) Broadband.



(d) Cybershake.

Fig. 5. Impact of scaling policies on resource utilization efficiency and application peformance for various workflows.

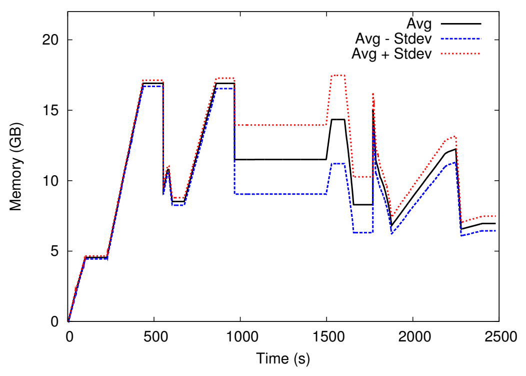


Fig. 6. Load balancing analysis for a run of Montage.

of partitions does not divide evenly to the number of nodes after two reconfigurations. Hence, a part of the nodes hold more partitions than the others. Thus, the system achieves this small load imbalance. The peaks/low points on the graph and then the sudden increase/decrease in the 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.

E. 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, thus also improving energy consumption. As expected, our results show that when increasing the policy aggressivity both the resource usage improvement and the application slowdown decrease. Our experiments show that with conservative scaling policies, MemEFS obtains a resource utilization improvement from 47% to 65.7% on all applications. The incurred performance overhead depends on the application structure, being at most 28% for three of the four evaluated workflows. In all these cases, *the performance overhead experienced by the user is much smaller than the resource utilization improvement*. The only workflow for which the performance overhead is compared to the resource utilization improvement is BLAST, which has a highly parallel structure. Nevertheless, these experiments show that users can choose from a space of different trade-offs, depending of their application structure and performance and utilization objectives.

V. Related Work

We discuss two classes of related work: (i) works that follow objectives similar to ours, focusing on elasticity and in-memory data storage; (ii) and works that address some of our design issues, focusing on hashing mechanisms for better load balancing in key-value stores.

A. 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 [25], GlusterFS [26], XtreemFS [27], HDFS [28], CEPH [29], 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 [30] and Lim et.al [31]. 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 [32] and FaRM [33] 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 [34], [35]. 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 [36] and Hazelcast [37] 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.

B. Load Balancing Schemes in DHTs

Several works focused on load balancing techniques for key-value stores [38], [39], [40], [16]. The most promising class of solutions is based on consistent hashing.

ZHT is a zero-hop distributed hash table [38]. 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 [39], [40], [16]: 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 [39] 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 sub-interval 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 [40]. 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 [16]. 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.

VI. Conclusion and Future Work

Scientific workflows exhibit significant storage demand variability during runtime. To overcome this issue, traditional approaches generally overprovision 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, based on the storage demand, while distributing data uniformly across system nodes to achieve a load balanced storage and network traffic.

We have shown that, with simple adaptation policies, MemEFS is capable to alleviate users from the need of estimating the application resource demands in advance and overprovisioning resources. Our experiments show that MemEFS is able to largely improve resource utilization while adding only modest performance overheads. Such results are a promising step in further exploring trade-offs between resource utilization, energy and application performance.

As future work we plan to explore the design space of scaling policies for MemEFS, to give users a range of options between application performance, used memory and monetary and energy consumption costs. We also plan to optimize the reconfiguration process of MemEFS, by allowing applications to continue their I/O operations during data migration. Finally, we plan to implement fault-tolerance in MemEFS.

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*, Vol. 4, no. 2, pp. 73–87, 2009.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, Vol. 215, no. 3, pp. 403–410, 1990.

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

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

[5] A. Uta, A. Sandu, and T. Kielmann, "Overcoming data locality: An in-memory runtime file system with symmetrical data distribution," *Future Generation Computer Systems*, 2015.

[6] ——, "MemFS: an In-Memory Runtime File System with Symmetrical Data Distribution," *IEEE Cluster*, 2014, pp. 272–273, (poster paper).

[7] S. Sanfilippo and P. Noordhuis, "Redis," http://redis.io, 2014.

[8] M. Szeredi *et al.*, "FUSE: Filesystem in userspace," *http://fuse.sourceforge.net/*, 2014.

[9] B. Aker, "Libmemcached," http://libmemcached.org/libMemcached.html, 2014.

[10] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux journal*, Vol. 2004, no. 124, p. 5, 2004.

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

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

[13] "xxhash," https://code.google.com/p/xxhash/, 2014.

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

[15] R. Rivest, "The md5 message-digest algorithm," 1992.

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

[17] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, Vol. 31, no. 4. ACM, 2001, pp. 149–160.

[18] "hiredis," https://github.com/redis/hiredis, 2014.

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

[20] "DAS-4, The Distributed ASCI Supercomputer," http://www.cs.vu.nl/das4/, 2014.

[21] "SCEC project, Southern California Earthquake Center," http://www.scec.org/, 2015.

[22] "Pegasus Workflow Generator," https://confluence.pegasus.isi.edu/display/ /pegasus/WorkflowGenerator, 2015.

[23] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, Vol. 29, no. 3, pp. 682–692, 2013.

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

[25] R. B. Ross, R. Thakur *et al.*, "PVFS: A parallel file system for linux clusters," *4th Annual Linux Showcase and Conference*, 2000, pp. 391–430.

[26] "GlusterFS," http://www.gluster.org/, 2014.

[27] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario, "The XtreemFS Architecture - A Case for Object-based File Systems in Grids," *Concurrency and computation: Practice and experience*, 2008.

[28] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[29] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," *7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.

[30] B. Nicolae, P. Riteau, and K. Keahey, "Bursting the Cloud Data Bubble: Towards Transparent Storage Elasticity in IaaS Clouds," *IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14, 2014, pp. 135–144.

[31] H. C. Lim, S. Babu, and J. S. Chase, "Automated control for elastic storage," *7th International Conference on Autonomic Computing*, ser. ICAC '10, 2010, pp. 1–10.

[32] 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*, Vol. 54, no. 7, pp. 121–130, 2011.

[33] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast Remote Memory," *11th USENIX Symposium on Networked Systems Design and Implementation*, 2014, pp. 401–414.

[34] N. S. Islam, X. Lu, M. Wasi-ur Rahman, R. Rajachandrasekar, and D. K. Panda, "In-memory i/o and replication for hdfs with memcached: Early experiences," *2014 IEEE International Conference on Big Data*. IEEE, 2014, pp. 213–218.

[35] F. R. Duro, J. G. Blas, and J. Carretero, "A hierarchical parallel storage system based on distributed memory for large scale systems," *20th European MPI Users' Group Meeting*, ser. EuroMPI '13. New York, NY, USA: ACM, 2013.

[36] "Amazon ElastiCache," http://aws.amazon.com/elasticache/, 2015.

[37] "Hazelcast," http://http://hazelcast.com/, 2015.

[38] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," *Parallel & Distributed Processing Symposium (IPDPS)*, 2013.

[39] A. Brinkmann, K. Salzwedel, and C. Scheideler, "Compact, Adaptive Placement Schemes for Non-uniform Requirements," *14th Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '02. New York, NY, USA: ACM, 2002, pp. 53–62.

[40] C. Schindelhauer and G. Schomaker, "Weighted Distributed Hash Tables," *ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '05. New York, NY, USA: ACM, 2005, pp. 218–227.