SpringFS: Bridging Agility and Performance in Elastic Distributed Storage

Lianghong Xu*, James Cipar*, Elie Krevat*, Alexey Tumanov*
Nitin Gupta*, Michael A. Kozuch†, Gregory R. Ganger*
*Carnegie Mellon University, †Intel Labs

Abstract

Elastic storage systems can be expanded or contracted to meet current demand, allowing servers to be turned off or used for other tasks. However, the usefulness of an elastic distributed storage system is limited by its agility: how quickly it can increase or decrease its number of servers. Due to the large amount of data they must migrate during elastic resizing, state-of-the-art designs usually have to make painful tradeoffs among performance, elasticity and agility.

This paper describes an elastic storage system, called SpringFS, that can quickly change its number of active servers, while retaining elasticity and performance goals. SpringFS uses a novel technique, termed *bounded write offloading*, that restricts the set of servers where writes to overloaded servers are redirected. This technique, combined with the read offloading and passive migration policies used in SpringFS, minimizes the work needed before deactivation or activation of servers. Analysis of real-world traces from Hadoop deployments at Facebook and various Cloudera customers and experiments with the SpringFS prototype confirm SpringFS's agility, show that it reduces the amount of data migrated for elastic resizing by up to two orders of magnitude, and show that it cuts the percentage of active servers required by 67–82%, outdoing state-of-the-art designs by 6–120%.

1 Introduction

Distributed storage can and should be elastic, just like other aspects of cloud computing. When storage is provided via single-purpose storage devices or servers, separated from compute activities, elasticity is useful for reducing energy usage, allowing temporarily unneeded storage components to be powered down. However, for storage provided via multi-purpose servers (e.g. when a server operates as both a storage node in a distributed filesystem and a compute node), such elasticity is even more valuable— providing cloud infrastructures with the freedom to use such servers for other purposes, as tenant demands and priorities dictate. This freedom may be particularly important for increasingly prevalent data-intensive computing activities (e.g., data analytics).

Data-intensive computing over big data sets is quickly becoming important in most domains and will be a major consumer of future cloud computing resources [7, 4, 3, 2]. Many of the frameworks for such computing (e.g., Hadoop [1] and Google's MapReduce [10]) achieve efficiency by distributing and storing the data on the same servers used for processing it. Usually, the data is replicated and spread evenly (via randomness) across the servers, and the entire set of servers is assumed to always be part of the data analytics cluster. Little-to-no support is provided for elastic sizing[1] of the portion of the cluster that hosts storage—only nodes that host no storage can be removed without significant effort, meaning that the storage service size can only grow.

Some recent distributed storage designs (e.g., Sierra [18], Rabbit [5]) provide for elastic sizing, originally targeted for energy savings, by distributing replicas among servers such that subsets of them can be powered down when the workload is low without affecting data availability; any server with the *primary replica* of data will remain active. These systems are designed mainly for performance or elasticity (how small the system size can shrink to) goals, while overlooking the importance of agility (how quickly the system can resize its footprint in response to workload variations), which we find has a significant impact on the machine-hour savings (and so the operating cost savings) one can potentially achieve. As a result, state-of-the-art elastic storage systems must make painful tradeoffs among these goals, unable to fulfill them at the same time. For example, Sierra balances load across all active servers and thus provides good performance. However, this even data layout limits elasticity— at least one third of the servers must always be active (assuming 3-way replication), wasting machine hours that could be used for other purposes when the workload is very low. Further, rebalancing the data layout when turning servers back on induces significant migration overhead, impairing system agility.

[1] We use "elastic sizing" to refer to dynamic online resizing, down from the full set of servers and back up, such as to adapt to workload variations. The ability to add new servers, as an infrequent administrative action, is common but does not itself make a storage service "elastic" in this context; likewise with the ability to survive failures of individual storage servers.

In contrast, Rabbit can shrink its active footprint to a much smaller size (≈10% of the cluster size), but its reliance on Everest-style write offloading [16] induces significant cleanup overhead when shrinking the active server set, resulting in poor agility.

This paper describes a new elastic distributed storage system, called SpringFS, that provides the elasticity of Rabbit *and* the peak write bandwidth characteristic of Sierra, while maximizing agility at each point along a continuum between their respective best cases. The key idea is to employ a small set of servers to store all primary replicas nominally, but (when needed) offload writes that would go to overloaded servers to only the *minimum* set of servers that can satisfy the write throughput requirement (instead of *all* active servers). This technique, termed *bounded write offloading,* effectively restricts the distribution of primary replicas during offloading and enables SpringFS to adapt dynamically to workload variations while meeting performance targets with a minimum loss of agility—most of the servers can be extracted without needing any pre-removal cleanup. SpringFS further improves agility by minimizing the cleanup work involved in resizing with two more techniques: *read offloading* offloads reads from write-heavy servers to reduce the amount of write offloading needed to achieve the system's performance targets; *passive migration* delays migration work by a certain time threshold during server re-integration to reduce the overall amount of data migrated. With these techniques, SpringFS achieves agile elasticity while providing performance comparable to a non-elastic storage system.

Our experiments demonstrate that the SpringFS design enables significant reductions in both the fraction of servers that need to be active and the amount of migration work required. Indeed, its design for where and when to offload writes enables SpringFS to resize elastically without performing any data migration at all in most cases. Analysis of traces from six real Hadoop deployments at Facebook and various Cloudera customers show the oft-noted workload variation and the potential of SpringFS to exploit it—SpringFS reduces the amount of data migrated for elastic resizing by up to two orders of magnitude, and cuts the percentage of active servers required by 67–82%, outdoing state-of-the-art designs like Sierra and Rabbit by 6–120%.

This paper makes three main contributions: First, to the best of our knowledge, it is the first to show the importance of agility in elastic distributed storage, highlighting the need to resize quickly (at times) rather than just hourly as in previous designs. Second, SpringFS introduces a novel write offloading policy that bounds the set of servers to which writes to over-loaded primary servers are redirected. Bounded write offloading, together with read offloading and passive migration significantly improve the system's agility by reducing the cleanup work during elastic resizing. These techniques apply generally to elastic storage with an uneven data layout. Third, we demonstrate the significant machine-hour savings that can be achieved with elastic resizing, using six real-world HDFS traces, and the effectiveness of SpringFS's policies at achieving a "close-to-ideal" machine-hour usage.

The remainder of this paper is organized as follows. Section 2 describes elastic distributed storage generally, the importance of agility in such storage, and the limitations of the state-of-the-art data layout designs in fulfilling elasticity, agility and performance goals at the same time. Section 3 describes the key techniques in SpringFS design and how they can increase agility of elasticity. Section 4 overviews the SpringFS implementation. Section 5 evaluates the SpringFS design.

2 Background and Motivation

This section motivates our work. First, it describes the related work on elastic distributed storage, which provides different mechanisms and data layouts to allow servers to be extracted while maintaining data availability. Second, it demonstrates the significant impact of agility on aggregate machine-hour usage of elastic storage. Third, it describes the limitations of state-of-the-art elastic storage systems and how SpringFS fills the significant gap between agility and performance.

2.1 Related Work

Most distributed storage is not elastic. For example, the cluster-based storage systems commonly used in support of cloud and data-intensive computing environments, such as the Google File System(GFS) [11] or the Hadoop Distributed Filesystem [1], use data layouts that are not amenable to elasticity. The Hadoop Distributed File System (HDFS), for example, uses a replication and data-layout policy wherein the first replica is placed on a node in the same rack as the writing node (preferably the writing node, if it contributes to DFS storage), the second and third on random nodes in a randomly chosen different rack than the writing node. In addition to load balancing, this data layout provides excellent availability properties—if the node with the primary replica fails, the other replicas maintain data availability; if an entire rack fails (e.g., through the failure of a communication link), data availability is maintained via the replica(s) in another rack. But, such a data layout prevents elasticity by requiring that almost all nodes be active—no more than one node per rack can be turned off without a high likelihood of making some data unavailable.

Recent research [5, 13, 18, 19, 17] has provided new data layouts and mechanisms for enabling elasticity in distributed storage. Most notable are Rabbit [5] and Sierra [18]. Both organize replicas such that one copy of data is always on a specific subset of servers, termed *primaries*, so as to allow the remainder of the nodes to be powered down without affecting availability, when the workload is low. With workload increase, they can be turned back on. The same designs and data distribution schemes would allow for servers to be used for other functions, rather than turned off, such as for higher-priority (or higher paying) tenants' activities. Writes intended for servers that are *inactive*[2] are instead written to other *active* servers—an action called *write availability offloading*—and then later reorganized (when servers become active) to conform to the desired data layout.

Rabbit and Sierra build on a number of techniques from previous systems, such as write availability offloading and power gears. Narayanan, Donnelly, and Rowstron [15] described the use of write availability offloading for power management in enterprise storage workloads. The approach was used to redirect traffic from otherwise idle disks to increase periods of idleness, allowing the disks to be spun down to save power. PARAID [20] introduced a geared scheme to allow individual disks in a RAID array to be turned off, allowing the power used by the array to be proportional to its throughput.

Everest [16] is a distributed storage design that used *write performance offloading*[3], rather than to avoid turning on powered-down servers, in the context of enterprise storage. In Everest, disks are grouped into distinct volumes, and each write is directed to a particular volume. When a volume becomes overloaded, writes can be temporarily redirected to other volumes that have spare bandwidth, leaving the overloaded volume to only handle reads. Rabbit applies this same approach, when necessary, to address overload of the primaries.

SpringFS borrows the ideas of write availability and performance offloading from prior elastic storage systems. We expand on previous work by developing new offloading and migration schemes that effectively eliminate the painful tradeoff between agility and write performance in state-of-the-art elastic storage designs.

[2]We generally refer to a server as *inactive* when it is either powered down or reused for other purposes. Conversely, we call a server *active* when it is powered on and servicing requests as part of a elastic distributed storage system.

[3]Write performance offloading differs from write availability offloading in that it offloads writes from overloaded *active* servers to other (relatively idle) *active* servers for better load balancing. The Everest-style and bounded write offloading schemes are both types of write performance offloading.

2.2 Agility is important

By "agility", we mean how quickly one can change the number of servers effectively contributing to a service. For most non-storage services, such changes can often be completed quickly, as the amount of state involved is small. For distributed storage, however, the state involved may be substantial. A storage server can service reads only for data that it stores, which affects the speed of both removing and re-integrating a server. Removing a server requires first ensuring that all data is available on other servers, and re-integrating a server involves replacing data overwritten (or discarded) while it was inactive.

The time required for such migrations has a direct impact on the machine-hours consumed by elastic storage systems. Systems with better agility are able to more effectively exploit the potential of workload variation by more closely tracking workload changes. Previous elastic storage systems rely on very infrequent changes (e.g., hourly resizing in Sierra [18]), but we find that over half of the potential savings is lost with such an approach due to the burstiness of real workloads.

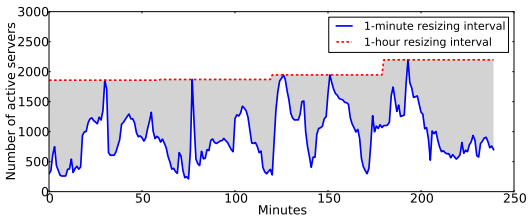

Figure 1: Workload variation in the Facebook trace. The shaded region represents the potential reduction in machine-hour usage with a 1-minute resizing interval.

As one concrete example, Figure 1 shows the number of active servers needed, as a function of time in the trace, to provide the required throughput in a randomly chosen 4-hour period from the Facebook trace described in Section 5. The dashed and solid curves bounding the shaded region represent the minimum number of active servers needed if using 1-hour and 1-minute resizing intervals, respectively. For each such period, the number of active servers corresponds to the number needed to provide the peak throughput in that period, as is done in Sierra to avoid significant latency increases. The area under each curve represents the machine time used for that resizing interval, and the shaded region represents the increased server usage (more than double) for the 1-hour interval. We observe similar burstiness and consequences of it across all of the traces.

2.3 Bridging Agility and Performance

Previous elastic storage systems overlook the importance of agility, focusing on performance and elasticity. This section describes the data layouts of state-of-the-art elastic storage systems, specifically Sierra and Rabbit, and how their layouts represent two specific points in the tradeoff space among elasticity, agility and performance. Doing so highlights the need for a more flexible elastic storage design that fills the void between them, providing greater agility and matching the best of each.

We focus on elastic storage systems that ensure data availability at all times. When servers are extracted from the system, at least one copy of all data must remain active to serve read requests. To do so, state-of-the-art elastic storage designs exploit data replicas (originally for fault tolerance) to ensure that all blocks are available at any power setting. For example, with 3-way replication[4], Sierra stores the first replica of every block (termed *primary replica*) in one third of servers, and writes the other 2 replicas to the other two thirds of servers. This data layout allows Sierra to achieve full peak performance due to balanced load across all active servers, but it limits the elasticity of the system by not allowing the system footprint to go below one third of the cluster size. We show in section 5.2 that such limitation can have a significant impact on the machine-hour savings that Sierra can potentially achieve, especially during periods of low workload.

Rabbit, on the other hand, is able to reduce its system footprint to a much smaller size ($\approx$10% of the cluster size). It does so by storing the replicas according to an *equal-work* data layout, so that it achieves *power proportionality* for read requests. That is, read performance scales linearly with the number of active servers: if 50% of the servers are active, the read performance of Rabbit should be at least 50% of its maximum read performance. The equal-work data layout ensures that, with any number of active servers, each server is able to perform an equal share of the read workload. In a system storing B blocks, with p primary servers and x active servers, each active server must store at least B/x blocks so that reads can be distributed equally, with the exception of the primary servers. Since a copy of all blocks must be stored on the p primary servers, they each store B/p blocks. This ensures (probabilistically) that when a large quantity of data is read, no server must read more than the others and become a bottleneck. This data layout allows Rabbit to keep the number of primary servers ($p = N/e^2$) very small (e is Euler's constant). The small number of primary servers provides great agility—Rabbit is able to shrink its system size down to p without any cleanup work—but it can create bottlenecks for writes. Since the primary servers must store the primary replicas for all blocks, the maximum write throughput of Rabbit is limited by the maximum aggregate write throughout of the p primary servers, even when all servers are active. In contrast, Sierra is able to achieve the same maximum write throughput as that of HDFS, that is, the aggregate write throughput of $N/3$ servers (recall: N servers write 3 replicas for every data block).

[4]We assume 3-way replication for all data blocks throughout this paper, which remains the default policy for HDFS. The data layout designs apply to other replication levels as well. Different approaches than Sierra, Rabbit and SpringFS are needed when erasure codes are used for fault tolerance instead of replication.

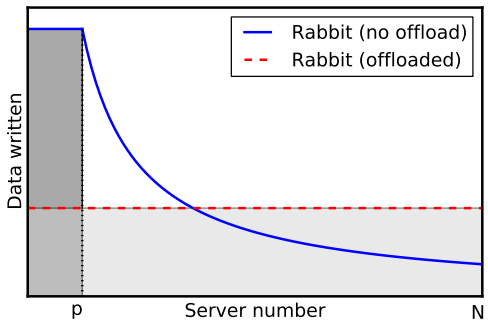

Figure 2: Primary data distribution for Rabbit without offloading (grey) and Rabbit with offloading (light grey). With offloading, primary replicas are spread across all active servers during writes, incurring significant cleanup overhead when the system shrinks its size.

Rabbit borrows write offloading from the Everest system [16] to solve this problem. When primary servers become the write performance bottleneck, Rabbit simply offloads writes that would go to heavily loaded servers across *all* active servers. While such write offloading allows Rabbit to achieve good peak write performance comparable to non-modified HDFS due to balanced load, it significantly impairs system agility by spreading primary replicas across all active servers, as depicted in Figure 2. Consequently, before Rabbit shrinks the system size, cleanup work is required to migrate some primary replicas to the remaining active servers so that at least one complete copy of data is still available after the resizing action. As a result, the improved performance from Everest-style write offloading comes at a high cost in system agility.

Figure 3 illustrates the very different design points represented by Sierra and Rabbit, in terms of the tradeoffs among agility, elasticity and peak write performance. Read performance is the same for all of these systems, given the same number of active servers. The number of servers that store primary replicas indicates the minimal system footprint one can shrink to without any cleanup work. As described above, state-of-the art elastic storage systems such as Sierra and Rabbit suffer from the painful tradeoff between agility and perfor-

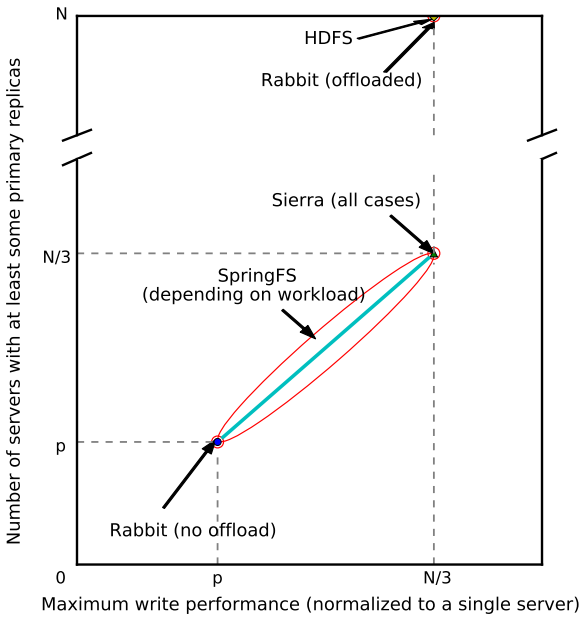

Figure 3: Elastic storage system comparison in terms of agility and performance. N is the total size of the cluster. p is the number of primary servers in the equal-work data layout. Servers with at least some primary replicas cannot be deactivated without first moving those primary replicas. SpringFS provides a continuum between Sierra's and Rabbit's (when no offload) single points in this tradeoff space. When Rabbit requires offload, SpringFS is superior at all points. Note that the y-axis is discontinuous.

mance due to the use of a rigid data layout. SpringFS provides a more flexible design that provides the best-case elasticity of Rabbit, the best-case write performance of Sierra, and much better agility than either. To achieve the range of options shown, SpringFS uses an explicit bound on the *offload set*, where writes of primary replicas to overloaded servers are offloaded to only the *minimum* set of servers (instead of *all* active servers) that can satisfy the current write throughput requirement. This additional degree of freedom allows SpringFS to adapt dynamically to workload changes, providing the desired performance while maintaining system agility.

3 SpringFS Design and Policies

This section describes SpringFS's data layout, as well as the bounded write offloading and read offloading policies that minimize the cleanup work needed before deactivation of servers. It also describes the passive migration policy used during a server's re-integration to address data that was written during the server's absence.

3.1 Data Layout and Offloading Policies

Data layout. Regardless of write performance, the equal-work data layout proposed in Rabbit enables the smallest number of primary servers and thus provides the best elasticity in state-of-the-art designs.[5] SpringFS retains such elasticity using a variant of the equal-work data layout, but addresses the agility issue incurred by Everest-style offloading when write performance bottlenecks arise. The key idea is to bound the distribution of primary replicas to a minimal set of servers (instead of offloading them to all active servers), given a target maximum write performance, so that the cleanup work during server extraction can be minimized. This *bounded write offloading* technique introduces a parameter called the *offload set*: the set of servers to which primary replicas are offloaded (and as a consequence receive the most write requests). The offload set provides an adjustable tradeoff between maximum write performance and cleanup work. With a small offload set, few writes will be offloaded, and little cleanup work will be subsequently required, but the maximum write performance will be limited. Conversely, a larger offload set will offload more writes, enabling higher maximum write performance at the cost of more cleanup work to be done later. Figure 4 shows the SpringFS data layout and its relationship with the state-of-the-art elastic data layout designs. We denote the size of the offload set as m, the number of primary servers in the equal-work layout as p, and the total size of the cluster as N. When m equals p, SpringFS behaves like Rabbit and writes all data according to the equal-work layout (no offload); when m equals $N/3$, SpringFS behaves like Sierra and load balances all writes (maximum performance). As illustrated in Figure 3, the use of the tunable offload set allows SpringFS to achieve both end points and points in between.

Choosing the offload set. The offload set is not a rigid setting, but determined on the fly to adapt to workload changes. Essentially, it is chosen according to the target maximum write performance identified for each resizing interval. Because servers in the offload set write one complete copy of the primary replicas, the size of the offload set is simply the maximum write throughput in the workload divided by the write throughput a single server can provide. Section 5.2 gives a more detailed description of how SpringFS chooses the offload set (and the number of active servers) given the target workload performance.

Read offloading. One way to reduce the amount of cleanup work is to simply reduce the amount of write offloading that needs to be done to achieve the system's

[5] Theoretically, no other data layout can achieve a smaller number of primary servers while maintaining power-proportionality for read performance.

performance targets. When applications simultaneously read and write data, SpringFS can coordinate the read and write requests so that reads are preferentially sent to higher numbered servers that naturally handle fewer write requests. We call this technique *read offloading*.

Despite its simplicity, read offloading allows SpringFS to increase write throughput without changing the offload set by taking read work away from the low numbered servers (which are the bottleneck for writes). When a read occurs, instead of randomly picking one among the servers storing the replicas, SpringFS chooses the server that has received the least number of total requests recently. (The one exception is when the client requesting the read has a local copy of the data. In this case, SpringFS reads the replica directly from that server to exploit machine locality.) As a result, lower numbered servers receive more writes while higher numbered servers handle more reads. Such read/write distribution balances the overall load across all the active servers while reducing the need for write offloading.

Replica placement. When a block write occurs, SpringFS chooses target servers for the 3 replicas in the following steps: The primary replica is load balanced across (and thus bounded in) the m servers in the current offload set. (The one exception is when the client requesting the write is in the offload set. In this case, SpringFS writes the primary copy to that server, instead of the server with the least load in the offload set, to exploit machine locality.) For non-primary replicas, SpringFS first determines their target servers according to the equal-work layout. For example, the target server for the secondary replica would be a server numbered between $p+1$ and ep, and that for the tertiary replica would be a server numbered between $ep+1$ and e^2p, both following the probability distribution as indicated by the equal-work layout (lower numbered servers have higher probability to write the non-primary replicas). If the target server number is higher than m, the replica is written to that server. However, if the target server number is between $p+1$ and m (a subset of the offload set), the replica is instead redirected and load balanced across servers outside the offload set, as shown in the shaded regions in Figure 4. Such redirection of non-primary replicas reduces the write requests going to the servers in the offload set and ensures that these servers store only the primary replicas.

Fault tolerance and multi-volume support. The use of an uneven data layout creates new problems for fault tolerance and capacity utilization. For example, when a primary server fails, the system may need to re-integrate some non-primary servers to restore the primary replicas onto a new server. SpringFS includes the data layout refinements from Rabbit that minimize the number of additional servers that must be re-activated if such failure happens. Writes that would have gone to the failed primary server are instead redirected to other servers in the offload set to preserve system agility. Like Rabbit, SpringFS also accommodates multi-volume data layouts in which independent volumes use distinct servers as primaries in order to allow small values of p without limiting storage capacity utilization to $3p/N$.

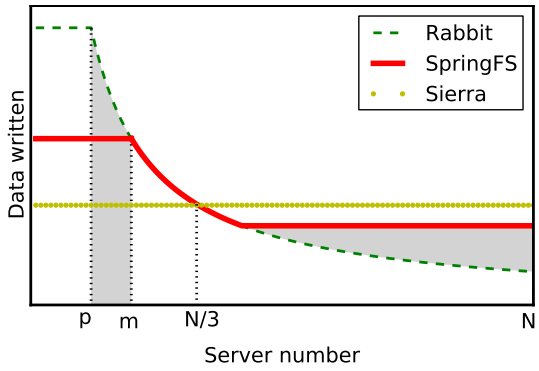

Figure 4: SpringFS data layout and its relationship with previous designs. The offload set allows SpringFS to achieve a dynamic tradeoff between the maximum write performance and the cleanup work needed before extracting servers. In SpringFS, all primary replicas are stored in the m servers of the offload set. The shaded regions indicate writes of non-primary replicas that would have gone to the offload set (in SpringFS) are instead redirected and load balanced outside the set.

3.2 Passive Migration for Re-integration

When SpringFS tries to write a replica according to its target data layout but the chosen server happens to be inactive, it must still maintain the specified replication factor for the block. To do this, another host must be selected to receive the write. *Availability offloading* is used to redirect writes that would have gone to inactive servers (which are unavailable to receive requests) to the active servers. As illustrated in Figure 5, SpringFS load balances availability offloaded writes together with the other writes to the system. This results in the availability offloaded writes going to the less-loaded active servers rather than adding to existing write bottlenecks on other servers.

Because of availability offloading, re-integrating a previously deactivated server is more than simply restarting its software. While the server can begin servicing its share of the write workload immediately, it can only service reads for blocks that it stores. Thus, filling it according to its place in the target equal-work layout is part of full re-integration.

When a server is reintegrated to address a workload

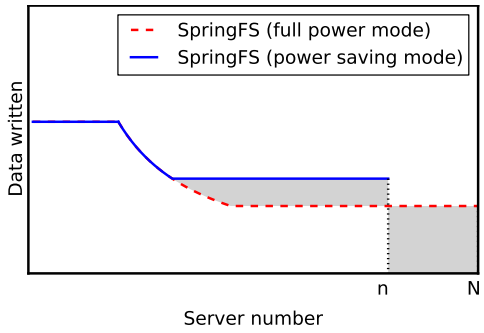

Figure 5: Availability offloading. When SpringFS works in the power saving mode, some servers ($n+1$ to N) are deactivated. The shaded regions show that writes that would have gone to these inactive servers are offloaded to higher numbered active servers for load balancing.

increase, the system needs to make sure that the active servers will be able to satisfy the read performance requirement. One option is to aggressively restore the equal work data layout before reintegrated servers begin servicing reads. We call this approach *aggressive migration*. Before anticipated workload increases, the migration agent would activate the right number of servers and migrate some data to the newly activated servers so that they store enough data to contribute their full share of read performance. The migration time is determined by the number of blocks that need to be migrated, the number of servers that are newly activated, and the I/O throughput of a single server. With aggressive migration, cleanup work is never delayed. Whenever a resizing action takes place, the property of the equal-work layout is obeyed—server x stores no less than $\frac{B}{x}$ blocks.

SpringFS takes an alternate approach called *passive migration*, based on the observation that cleanup work when re-integrating a server is not as important as when deactivating a server (for which it preserves data availability), and that the total amount of cleanup work can be reduced by delaying some fraction of migration work while performance goals are still maintained (which makes this approach better than aggressive migration). Instead of aggressively fixing the data layout (by activating the target number of servers in advance for a longer period of time), SpringFS temporarily activates more servers than would minimally be needed to satisfy the read throughput requirement and utilizes the extra bandwidth for migration work and to address the reduced number of blocks initially on each reactivated server. The number of extra servers that need to be activated is determined in two steps. First, an initial number is chosen to ensure that the number of valid data blocks still stored on the activated servers is more than the fraction of read workload they need to satisfy, so that the performance requirement is satisfied. Second, the number may be increased so that the extra servers provide enough I/O bandwidth to finish a fraction ($1/T$, where T is the migration threshold as described below) of migration work. To avoid migration work building up indefinitely, the migration agent sets a time threshold so that whenever a migration action takes place, it is guaranteed to finish within T minutes. With $T > 1$ (the default resizing interval), SpringFS delays part of the migration work while satisfying throughput requirement. Because higher numbered servers receive more writes than their equal-work share, due to write offloading, some delayed migration work can be replaced by future writes, which reduces the overall amount of data migration. If T is too large, however, the cleanup work can build up so quickly that even activating all the servers cannot satisfy the throughput requirement. In practice, we find a migration threshold $T = 10$ to be a good choice and use this setting for the trace analysis in Section 5. Exploring automatic setting of T is an interesting future work.

4 Implementation

SpringFS is implemented as a modified instance of the Hadoop Distributed File System (HDFS), version 0.19.1[6]. We build on a *Scriptable Hadoop* interface that we built into Hadoop to allow experimenters to implement policies in external programs that are called by the modified Hadoop. This enables rapid prototyping of new policies for data placement, read load balancing, task scheduling, and re-balancing. It also enables us to emulate both Rabbit and SpringFS in the same system, for better comparison. SpringFS mainly consists of four components: data placement agent, load balancer, resizing agent and migration agent, all implemented as python programs called by the Scriptable Hadoop interface.

Data placement agent. The data placement agent determines where to place blocks according to the SpringFS data layout. Ordinarily, when a HDFS client wishes to write a block, it contacts the HDFS NameNode and asks where the block should be placed. The NameNode returns a list of pseudo-randomly chosen DataNodes to the client, and the client writes the data directly to these DataNodes. The data placement agent starts together with the NameNode, and communicates with the NameNode using a simple text-based protocol over `stdin` and `stdout`. To obtain a placement decision for the R replicas of a block, the NameNode writes the name of the client machine as well as a list of candi-

[6]0.19.1 was the latest Hadoop version when our work started. We have done a set of experiments to verify that HDFS performance differs little, on our experimental setup, between version 0.19.1 and the latest stable version (1.2.1). We believe our results and findings are not significantly affected by still using this older version of HDFS.

date DataNodes to the placement agent's `stdin`. The placement agent can then filter and reorder the candidates, returning a prioritized list of targets for the write operation. The NameNode then instructs the client to write to the first R candidates returned.

Load balancer. The load balancer implements the read offloading policy and preferentially sends reads to higher numbered servers that handle fewer write requests whenever possible. It keeps an estimate of the load on each server by counting the number of requests sent to each server recently. Every time SpringFS assigns a block to a server, it increments a counter for the server. To ensure that recent activity has precedence, these counters are periodically decayed by 0.95 every 5 seconds. While this does not give the exact load on each server, we find its estimates good enough (within 3% off optimal) for load balancing among relatively homogeneous servers.

Resizing agent. The resizing agent changes SpringFS's footprint by setting an *activity state* for each DataNode. On every read and write, the data placement agent and load balancer will check these states and remove all "INACTIVE" DataNodes from the candidate list. Only "ACTIVE" DataNodes are able to service reads or writes. By setting the activity state for DataNodes, we allow the resources (e.g., CPU and network) of "INACTIVE" nodes to be used for other activities with no interference from SpringFS activities. We also modified the HDFS mechanisms for detecting and repairing under-replication to assume that "INACTIVE" nodes are not failed, so as to avoid undesired re-replication.

Migration agent. The migration agent crawls the entire HDFS block distribution (once) when the NameNode starts, and it keeps this information up-to-date by modifying HDFS to provide an interface to get and change the current data layout. It exports two metadata tables from the NameNode, mapping file names to block lists and blocks to DataNode lists, and loads them into a SQLite database. Any changes to the metadata (e.g., creating a file, creating or migrating a block) are then reflected in the database on the fly. When data migration is scheduled, the SpringFS migration agent executes a series of SQL queries to detect layout problems, such as blocks with no primary replica or hosts storing too little data. It then constructs a list of migration actions to repair these problems. After constructing the full list of actions, the migration agent executes them in the background. To allow block-level migration, we modified the HDFS client utility to have a "relocate" operation that copies a block to a new server. The migration agent uses GNU Parallel to execute many relocates simultaneously.

5 Evaluation

This section evaluates SpringFS and its offloading policies. Measurements of the SpringFS implementation show that it provide performance comparable to unmodified HDFS, that its policies improve agility by reducing the cleanup required, and that it can agilely adapt its number of active servers to provide required performance levels. In addition, analysis of six traces from real Hadoop deployments shows that SpringFS's agility enables significantly reduced commitment of active servers for the highly dynamic demands commonly seen in practice.

5.1 SpringFS prototype experiments

Experimental setup: Our experiments were run on a cluster of 31 machines. The modified Hadoop software is run within KVM virtual machines, for software management purposes, but each VM gets its entire machine and is configured to use all 8 CPU cores, all 8 GB RAM, and 100 GB of local hard disk space. One machine was configured as the Hadoop master, hosting both the NameNode and the JobTracker. The other 30 machines were configured as slaves, each serving as an HDFS DataNode and a Hadoop TaskTracker. Unless otherwise noted, SpringFS was configured for 3-way replication ($R = 3$) and 4 primary servers ($p = 4$).

To simulate periods of high I/O activity, and effectively evaluate SpringFS under different mixes of I/O operations, we used a modified version of the standard Hadoop TestDFSIO storage system benchmark called TestDFSIO2. Our modifications allow for each node to generate a mix of block-size (128 MB) reads and writes, distributed randomly across the block ID space, with a user-specified write ratio.

Except where otherwise noted, we specify a file size of 2GB per node in our experiments, such that the single Hadoop map task per node reads or writes 16 blocks. The total time taken to transfer all blocks is aggregated and used to determine a global throughput. In some cases, we break down the throughput results into the average aggregate throughput of just the block reads or just the block writes. This enables comparison of SpringFS's performance to the unmodified HDFS setup with the same resources. Our experiments are focused primarily on the relative performance changes as agility-specific parameters and policies are modified. Because the original Hadoop implementation is unable to deliver the full performance of the underlying hardware, our system can only be compared reasonably with it and not the capability of the raw storage devices.

Effect of offloading policies: Our evaluation focuses on how SpringFS's offloading policies affect per-

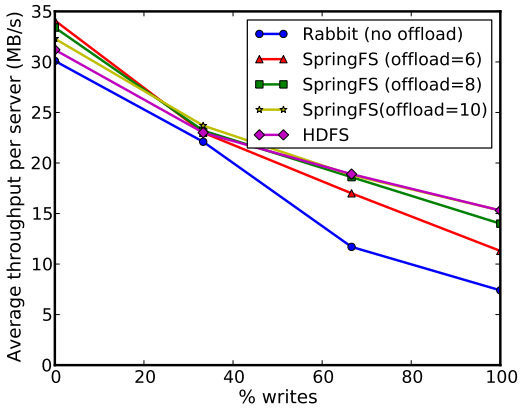

Figure 6: Performance comparison of Rabbit with no offload, original HDFS, and SpringFS with varied offload set.

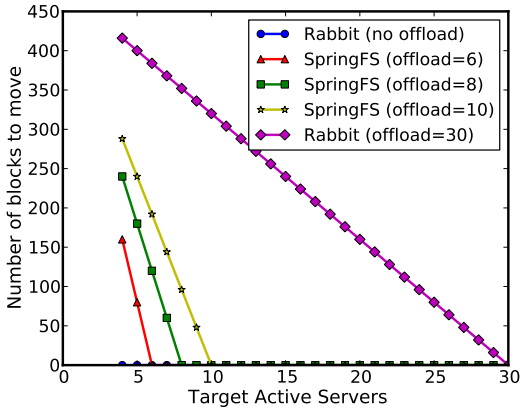

Figure 7: Cleanup work (in blocks) needed to reduce active server count from 30 to X, for different offload settings. The "(offload=6)", "(offload=8)" and "(offload=10)" lines correspond to SpringFS with bounded write offloading. The "(offload=30)" line corresponds to Rabbit using Everest-style write offloading. Deactivating only non-offload servers requires no block migration. The amount of cleanup work is linear in the number of target active servers.

formance and agility. We also measure the cleanup work created by offloading and demonstrate that SpringFS's number of active servers can be adapted agilely to changes in workload intensity, allowing machines to be extracted and used for other activities.

Figure 6 presents the peak sustained I/O bandwidth measured for HDFS, Rabbit and SpringFS at different offload settings. (Rabbit and SpringFS are identical when no offloading is used.) In this experiment, the write ratio is varied to demonstrate different mixes of read and write requests. SpringFS, Rabbit and HDFS achieve similar performance for a read-only workload, because in all cases there is a good distribution of blocks and replicas across the cluster over which to balance the load. The read performance of SpringFS slightly outperforms the original HDFS due to its explicit load tracking for balancing.

When no offloading is needed, both Rabbit and SpringFS are highly elastic and able to shrink 87% (26 non-primary servers out of 30) with no cleanup work. However, as the write workload increases, the equal-work layout's requirement that one replica be written to the primary set creates a bottleneck and eventually a slowdown of around 50% relative to HDFS for a maximum-speed write-only workload. SpringFS provides the flexibility to tradeoff some amount of agility for better write throughput under periods of high write load. As the write ratio increases, the effect of SpringFS's offloading policies becomes more visible. Using only a small number of offload servers, SpringFS significantly reduces the amount of data written to the primary servers and, as a result, significantly improves performance over Rabbit. For example, increasing the offload set from four (i.e., just the four primaries) to eight doubles maximum throughput for the write-only workload, while remaining agile—the cluster is still able to shrink 74% with no cleanup work.

Figure 7 shows the number of blocks that need to be relocated to preserve data availability when reducing the number of active servers. As desired, SpringFS's data placements are highly amenable to fast extraction of servers. Shrinking the number of nodes to a count exceeding the cardinality of the offload set requires no clean-up work. Decreasing the count into the write offload set is also possible, but comes at some cost. As expected, for a specified target, the cleanup work grows with an increase in the offload target set. SpringFS with no offload reduces to the based equal-work layout, which needs no cleanup work when extracting servers but suffers from write performance bottlenecks. The most interesting comparison is Rabbit's full offload (offload=30) against SpringFS's full offload (offload=10). Both provide the cluster's full aggregate write bandwidth, but SpringFS's offloading scheme does it with much greater agility—66% of the cluster could still be extracted with no cleanup work and more with small amounts of cleanup. We also measured actual cleanup times, finding (not surprisingly) that they correlate strongly with the number of blocks that must be moved.

SpringFS's read offloading policy is simple and reduces the cleanup work resulting from write offloading. To ensure that its simplicity does not result in lost opportunity, we compare it to the optimal, oracular scheduling policy with claircognizance of the HDFS layout. We

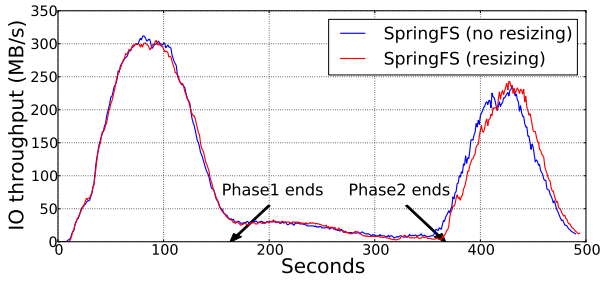

Figure 8: Agile resizing in a 3-phase workload

use an Integer Linear Programming (ILP) model that minimizes the number of reads sent to primary servers from which primary replica writes are offloaded. The SpringFS read offloading policy, despite its simple realization, compares favorably and falls within 3% from optimal on average.

Agile resizing in SpringFS: Figure 8 illustrates SpringFS's ability to resize quickly and deliver required performance levels. It uses a sequence of three benchmarks to create phases of workload intensity and measures performance for two cases: "SpringFS (no resizing)" where the full cluster stays active throughout the experiment and "SpringFS (resizing)" where the system size is changed with workload intensity. As expected, the performance is essentially the same for the two cases, with a small delay observed when SpringFS re-integrates servers for the third phase. But, the number of machine hours used is very different, as SpringFS extracts machines during the middle phase.

This experiment uses a smaller setup, with only 7 DataNodes, 2 primaries, 3 in the offload set, and 2-way replication. The workload consists of 3 consecutive benchmarks. The first benchmark is a TestDFSIO2 benchmark that writes 7 files, each 2GB in size for a total of 14GB written. The second benchmark is one SWIM job [9] randomly picked from a series of SWIM jobs synthesized from a Facebook trace which reads 4.2GB and writes 8.4GB of data. The third benchmark is also a TestDFSIO2 benchmark, but with a write ratio of 20%. The TestDFSIO2 benchmarks are I/O intensive, whereas the SWIM job consumes only a small amount of the full I/O throughput. For the resizing case, 4 servers are extracted after the first write-only TestDFSIO2 benchmark finishes (shrinking the active set to 3), and those servers are reintegrated when the second TestDFSIO2 job starts. In this experiment, the resizing points are manually set when phase switch happens. Automatic resizing can be done based on previous work on workload prediction [6, 12, 14].

The results in Figure 8 are an average of 10 runs for both cases, shown with a moving average of 3 seconds. The I/O throughput is calculated by summing read throughput and write throughput multiplied by the replication factor. Decreasing the number of active SpringFS servers from 7 to 3 does not have an impact on its performance, since no cleanup work is needed. As expected, resizing the cluster from 3 nodes to 7 imposes a small performance overhead due to background block migration, but the number of blocks to be migrated is very small—about 200 blocks are written to SpringFS with only 3 active servers, but only 4 blocks need to be migrated to restore the equal-work layout. SpringFS's offloading policies keep the cleanup work small, for both directions. As a result, SpringFS extracts and re-integrates servers very quickly.

5.2 Policy analysis with real-world traces

This subsection evaluates SpringFS in terms of machine-hour usage with real-world traces from six industry Hadoop deployments and compares it against three other storage systems: Rabbit, Sierra, and the default HDFS. We evaluate each system's layout policies with each trace, calculate the amount of cleanup work and the estimated cleaning time for each resizing action, and summarize the aggregated machine-hour usage consumed by each system for each trace. The results show that SpringFS significantly reduces machine-hour usage even compared to the state-of-the-art elastic storage systems, especially for write-intensive workloads.

Trace overview: We use traces from six real Hadoop deployments representing a broad range of business activities, one from Facebook and five from different Cloudera customers. The six traces are described and analyzed in detail by Chen et al. [8]. Table 1 summarizes key statistics of the traces. The Facebook trace (FB) comes from Hadoop DataNode logs, each record containing timestamp, operation type (HDFS_READ or HDFS_WRITE), and the number of bytes processed. From this information, we calculate the aggregate HDFS read/write throughput as well as the total throughput, which is the sum of read and write throughput multiplied by the replication factor (3 for all the traces). The five Cloudera customer traces (CC-a through CC-e, using the terminology from [8]) all come from Hadoop job history logs, which contain per-job records of job duration, HDFS input/output size, etc. Assuming the amount of HDFS data read or written for each job is distributed evenly within the job duration, we also obtain the aggregated HDFS throughput at any given point of time, which is then used as input to the analysis program.

Trace analysis and results: To simplify calculation, we make several assumptions. First, the maximum measured total throughput in the traces corresponds to the maximum aggregate performance across all the machines in the cluster. Second, the maximum throughput

Table 1: Trace summary. CC is "Cloudera Customer" and FB is "Facebook". HDFS bytes processed is the sum of HDFS bytes read and HDFS bytes written.

Trace	Machines	Date	Length	Bytes processed
CC-a	<100	2011	1 month	69TB
CC-b	300	2011	9 days	473TB
CC-c	700	2011	1 month	13PB
CC-d	400-500	2011	2.8 months	5PB
CC-e	100	2011	9 days	446TB
FB	3000	2010	10 days	10.5PB

a single machine can deliver, not differentiating reads and writes, is derived from the maximum measured total throughput divided by the number of machines in the cluster. In order to calculate the machine hour usage for each storage system, the analysis program needs to determine the number of active servers needed at any given point of time. It does this in the following steps: First, it determines the number of active servers needed in the imaginary "ideal" case, where no cleanup work is required at all, by dividing the total HDFS throughput by the maximum throughput a single machine can deliver. Second, it iterates through the number of active servers as a function of time. For each decrease in the active set of servers, it checks for any cleanup work that must be done by analyzing the data layout at that point. If any cleanup is required, it delays resizing until the work is done or the performance requirement demands an increase of the active set, to allow additional bandwidth for necessary cleanup work. For increases in the active set of servers, it turns on some extra servers to satisfy the read throughput and uses the extra bandwidth to do a fraction of migration work, using the passive migration policy (for all the systems) with the migration threshold set to be T=10.

Figures 9 and 10 show the number of active servers needed, as a function of time, for the 6 traces. Each graph has 4 lines, corresponding to the "ideal" storage system, SpringFS, Rabbit and Sierra, respectively. We

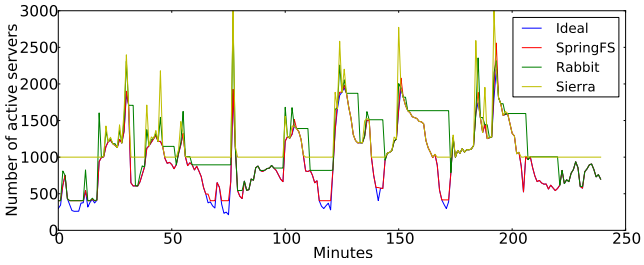

Figure 9: Facebook trace

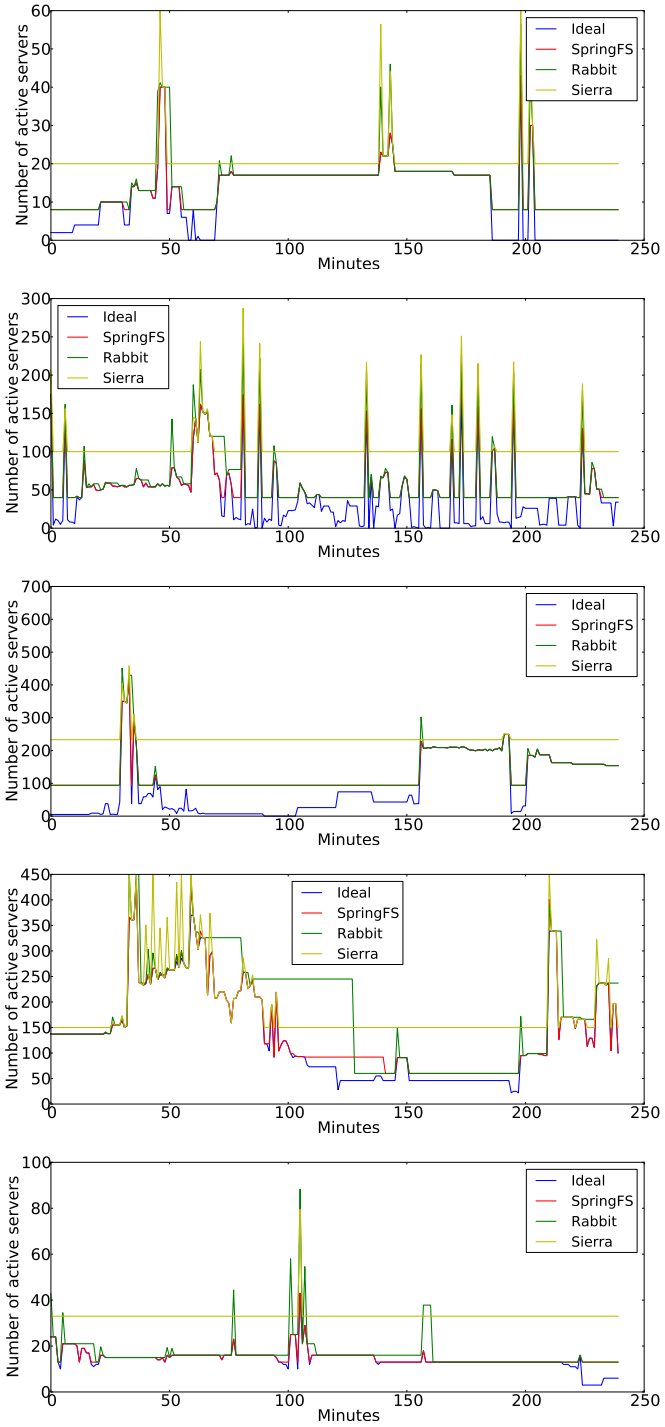

Figure 10: Traces: CC-a, CC-b, CC-c, CC-d, and CC-e

do not show the line for the Default HDFS, but since it is not elastic, its curve would be a horizontal line with the number of active servers always being the full cluster size (the highest value on the Y axis). While the original trace durations range from 9 days to 2.8 months, we only show a 4-hour-period for each trace for clarity. We start trace

replaying more than 3 days before the 4-hour period, to make sure it represents the situation when systems are in a steady state and includes the effect of delaying migration work.

As expected, SpringFS exhibits better agility than Rabbit, especially when shrinking the size of the cluster, since it needs no cleanup work until resizing down to the offload set. Such agility difference between SpringFS and Rabbit is shown in Figure 9 at various points of time (e.g., at minute 110, 140, and 160). The gap between the two lines indicates the number of machine hours saved due to the agility-aware read and bounded write policies used in SpringFS. SpringFS also achieves lower machine-hour usage than Sierra, as confirmed in all the analysis graphs. While a Sierra cluster can shrink down to 1/3 of its total size without any cleanup work, it is not able to further decrease the cluster size. In contrast, SpringFS can shrink the cluster size down to approximately 10% of the original footprint. When I/O activity is low, the difference in minimal system footprint can have a significant impact on the machine-hour usage (e.g., as illustrated in Figure 10(b), Figure 10(c) and Figure 10(e)). In addition, when expanding cluster size, Sierra incurs more cleaning overhead than SpringFS, because deactivated servers need to migrate more data to restore its even data layout. These results are summarized in Figure 11, which shows the extra number of machine hours used by each storage system, compared and normalized to the ideal system. In these traces, SpringFS outperforms the other systems by 6% to 120%. For the traces with a relatively high write ratio, such as the FB, CC-d and CC-e traces, SpringFS is able to achieve a "close-to-ideal" (within 5%) machine-hour usage. SpringFS is less close to ideal for the other three traces because they frequently need even less than the 13% primary servers that SpringFS cannot deactivate.

Figure 12 summarizes the total amount of data migrated by Rabbit, Sierra and SpringFS while running each trace. With bounded write offloading and read offloading, SpringFS is able to reduce the amount of data migration by a factor of 9–208, as compared to Rabbit. SpringFS migrates significantly less data than Sierra as well, because data migrated to restore the equal-work data layout is much less than that to restore an even data layout.

All of the trace analysis above assumes passive migration during server reintegration for all three systems compared, since it is useful to all of them. To evaluate the advantage of passive migration, specifically, we repeated the same trace analysis using the aggressive migration policy. The results show that passive migration reduces the amount of data migrated, relative to aggressive migration, by 1.5–7× (across the six traces) for SpringFS, 1.2–5.6× for Sierra, and 1.2–3× for Rabbit. The benefit for Sierra and SpringFS is more significant, because their data migration occurs primarily during server reintegration.

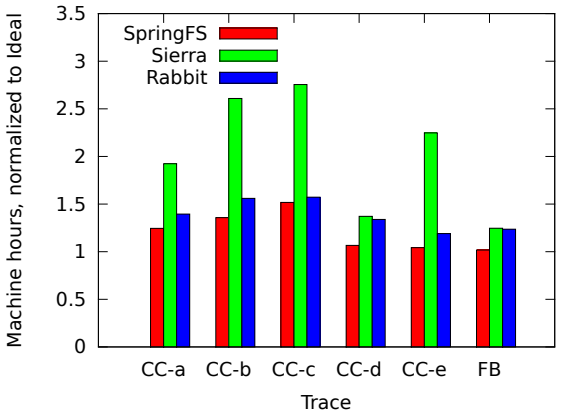

Figure 11: Number of machine hours needed to execute each trace for each system, normalized to the "Ideal" system (1 on the y-axis, not shown).

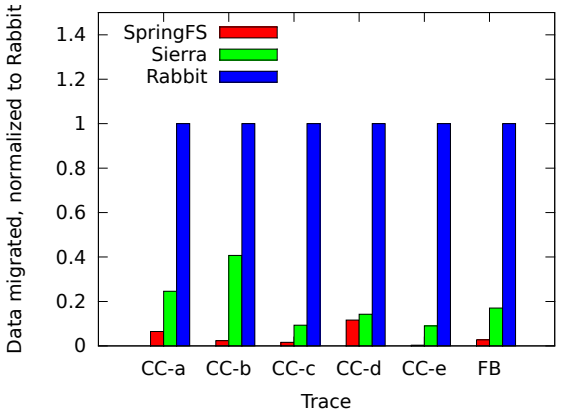

Figure 12: Total data migrated for Rabbit, Sierra and SpringFS, normalized to results for Rabbit.

6 Conclusion

SpringFS is a new elastic storage system that fills the space between state-of-the-art designs in the tradeoff among agility, elasticity, and performance. SpringFS's data layout and offloading/migration policies adapt to workload demands and minimize the data redistribution cleanup work needed for elastic resizing, greatly increasing agility relative to the best previous elastic storage designs. As a result, SpringFS can satisfy the time-varying performance demands of real environments with many fewer machine hours. Such agility provides an important building block for resource-efficient data-intensive

computing (a.k.a. Big Data) in multi-purpose clouds with competing demands for server resources.

There are several directions for interesting future work. For example, the SpringFS data layout assumes that servers are approximately homogeneous, like HDFS does, but some real-world deployments end up with heterogeneous servers (in terms of I/O throughput and capacity) as servers are added and replaced over time. The data layout could be refined to exploit such heterogeneity, such as by using more powerful servers as primaries. Second, SpringFS's design assumes a relatively even popularity of data within a given dataset, as exists for Hadoop jobs processing that dataset, so it will be interesting to explore what aspects change when addressing the unbalanced access patterns (e.g., Zipf distribution) common in servers hosting large numbers of relatively independent files.

Acknowledgements: We thank Cloudera and Facebook for sharing the traces and Yanpei Chen for releasing SWIM. We thank the members and companies of the PDL Consortium (including Actifio, APC, EMC, Facebook, Fusion-io, Google, HP Labs, Hitachi, Huawei, Intel, Microsoft Research, NEC Labs, NetApp, Oracle, Panasas, Samsung, Seagate, Symantec, VMWare, and Western Digital) for their interest, insights, feedback, and support. This research was sponsored in part by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC). Experiments were enabled by generous hardware donations from Intel, NetApp, and APC.

References

[1] Hadoop, 2012. http://hadoop.apache.org.

[2] MIT, Intel unveil new initiatives addressing 'Big Data', 2012. http://web.mit.edu/newsoffice/2012/big-data-csail-intel-center-0531.html.

[3] AMPLab, 2013. http://amplab.cs.berkeley.edu.

[4] ISTC-CC Research, 2013. www.istc-cc.cmu.edu.

[5] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan. Robust and flexible power-proportional storage. *ACM Symposium on Cloud Computing*, pages 217–228, 2010.

[6] P. Bodik, M. Armbrust, K. Canini, A. Fox, M. Jordan, and D. Patterson. A case for adaptive datacenters to conserve energy and improve reliability. *University of California at Berkeley, Tech. Rep. UCB/EECS-2008-127*, 2008.

[7] R. E. Bryant. Data-Intensive Supercomputing: The Case for DISC. Technical report, Carnegie Mellon University, 2007.

[8] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross industry study of mapreduce workloads. *VLDB*, 2012.

[9] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. *MASCOTS*, 2011.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–108, January 2008.

[11] S. Ghemawat, H. Gobioff, and S. tak Leung. The Google File System. In *SOSP*, pages 29–43, 2003.

[12] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Workload analysis and demand prediction of enterprise data center applications. *IISWC*, 2007.

[13] J. Leverich and C. Kozyrakis. On the Energy (In)efficiency of Hadoop Clusters. *HotPower*, 2009.

[14] M. Lin, A. Wierman, L. L. Andrew, and E. Thereska. Dynamic right-sizing for power-proportional data centers. *INFOCOM*, 2011.

[15] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *USENIX Conference on File and Storage Technologies*, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

[16] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through I/O off-loading. In *OSDI*, 2008.

[17] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: Building Distributed Enterprise Disk Arrays from Commodity Components. In *ASPLOS*, pages 48–58, 2004.

[18] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *EuroSys*, pages 169–182, 2011.

[19] N. Vasić, M. Barisits, V. Salzgeber, and D. Kostic. Making Cluster Applications Energy-Aware. In *Workshop on Automated Control for Datacenters and Clouds*, pages 37–42, New York, 2009.

[20] C. Weddle, M. Oldham, J. Qian, A.-I. A. Wang, P. L. Reiher, and G. H. Kuenning. PARAID: A Gear-Shifting Power-Aware RAID. *TOS*, 2007.