JackRabbit: Improved agility in elastic distributed storage

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

CMU-PDL-12-112
October 2012

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

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. This paper describes an elastic storage system, called JackRabbit, that can quickly change its number of active servers. JackRabbit uses a combination of agility-aware offloading and reorganization techniques to minimize the work needed before deactivation or activation of servers. Analysis of real-world traces and experiments with the JackRabbit prototype confirm JackRabbit's agility and show that it significantly reduces wasted server-time relative to state-of-the-art designs.

Acknowledgements: We thank the members and companies of the PDL Consortium (including Actifio, APC, EMC, Emulex, Facebook, Fusion-io, Google, Hewlett-Packard Labs, Hitachi, Huawei Technologies Co., Intel, Microsoft Research, NEC Laboratories, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, VMWare, and Western Digital) for their interest, insights, feedback, and support. This research was sponsored in part by Intel and by CyLab at Carnegie Mellon University under grant #DAAD19–02–1–0389 from the Army Research Office. Experiments were enabled by generous hardware donations from Intel, NetApp, and APC. We thank Cloudera and Facebook for sharing the traces and Yanpei Chen for releasing SWIM.

Keywords: cloud storage, distributed file systems

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. For storage provided via multi-purpose servers, however, such elasticity is needed to provide the cloud infrastructure with the freedom to use those servers for other purposes, when tenant demands and priorities dictate such usage. 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 [8, 3, 2, 5]. Many of the frameworks for such computing (e.g., Hadoop [1] and Google's MapReduce [12]) 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, 19], Rabbit [6]) provide for elastic sizing, originally targeted for energy savings, by distributing replicas among servers such that subsets of them can be powered down without affecting data 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. That is, a subset of the *active* servers can be *extracted* temporarily from such elastic storage services (whether to be powered down or used for other work) and then later *re-integrated* when needed. These designs assume that (1) the set of active servers contains at least one replica of all data at all times– providing data availability, and (2) data on extracted servers are durable– maintaining a level of fault-tolerance comparable to non-elastic systems[2].

Agility is an important metric for making such elasticity useful. By "agility", we mean how quickly one can change the number of servers effectively contributing to the service. For most non-storage services, such changes can often be completed quickly, as the amount of state involved is not large. For distributed storage, the state involved is substantial. A distributed 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 reintegration of a server involves replacing data overwritten (or discarded) while the server was not part of the storage service.

This paper describes a new elastic distributed storage system, called JackRabbit, focusing on its new policies designed to maximize the agility of elastic storage while accommodating performance and fault tolerance goals. JackRabbit builds on one of the recent (state-of-the-art) elastic storage designs—the Rabbit design mentioned above—which provides for elastic sizing down to a small percentage ($\approx$10%) of the cluster size without sacrificing balanced load. JackRabbit introduces new policies for data placement, workload distribution, and re-population of re-integrated servers. For example, server removal can be significantly delayed by the write offloading used to overcome overload of servers holding primary replicas; to address this, JackRabbit's offloading policies maximize the number of servers that can be extracted without needing any pre-removal redistribution, by offloading reads instead of writes whenever possible and bounding the set

[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.

[2]Failures of active servers may induce temporary windows of data unavailability; these will last until extracted servers storing the necessary replicas are re-integrated.

of servers used for write offloading.

Our experiments demonstrate that JackRabbit retains Rabbit's elasticity while providing improved agility, enabling significant reductions in the fraction of servers that need to be active and the amount of redistribution work involved. Indeed, its policies for where and when to offload writes allow JackRabbit to elastically resize itself without performing any data migration at all. Analyses of traces from six real Hadoop deployments at Facebook and various Cloudera customers show the oft-noted workload variation and the potential of JackRabbit's policies to exploit it—JackRabbit's policies reduce the percentage of active servers required by 58–80%, outdoing state-of-the-art designs like Sierra and Rabbit by 8–67%.

The remainder of this paper is organized as follows. Section 2 describes elastic distributed storage generally, the state-of-the-art Rabbit architecture for such storage, and agility issues that must be addressed. Section 3 describes the JackRabbit policies and how they can increase agility of elasticity. Section 4 overviews the JackRabbit implementation used for experiments. Section 5 evaluates the JackRabbit policies.

2 Elastic distributed storage

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) [13] 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 has provided new data layouts and mechanisms for enabling elasticity in distributed storage. This section describes this related work, additional details about one specific instance (Rabbit), and the issues that impede agility of elasticity for distributed storage. The next section describes our new policies for addressing those issues.

2.1 Related work

The primary previous work related to elastic distributed storage focuses on energy savings. Specifically, several recent research efforts [6, 14, 19, 20, 17] describe different mechanisms and data layouts to allow servers to be turned off while maintaining availability.

Most notable are Rabbit [6], discussed in the next subsection, and Sierra [18, 19]. 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. Writes directed at powered-down servers are instead written to other servers—an action called "write offloading" [15]—and then later reorganized (when servers are turned on) to conform to the desired data layout. Sierra's and Rabbit's offloading mechanisms are similar, differing in two ways. First, the data layout used by Sierra is fixed; a write directed at server A must eventually reach server A, while in Rabbit the final layout is determined at re-balancing time. Second, when a server is not active, Sierra's write offloading agent picks a fixed set of servers to handle the offloaded data, while in Rabbit the offloading targets are chosen on a per-block basis.

Rabbit and Sierra build on a number of techniques from previous systems, such as write offloading and power gears. Narayanan, Donnelly, and Rowstron [15] described the use of write offloading for power management in enterprise storage workloads. Write offloading 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 [21] 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 offloading for performance, 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.

We expand on previous work on elastic distributed storage by developing new read and write offloading mechanisms that mitigate agility limitations in prior designs. We build on the Rabbit design, because that system was made available to us, but the new policies apply generally to elastic storage.

2.2 Rabbit data distribution policies

Rabbit [6] is a distributed file system designed to provide power proportionality on workloads. It is based on HDFS [7], but uses alternate data layouts that allow it to extract (e.g., power off) large subsets of servers without reducing data availability. To do so, Rabbit exploits the data replicas (originally for fault tolerance) to ensure that all blocks are available at any power setting. In addition to maintaining availability, Rabbit tries to be *power proportional* with any number of active servers. That is, performance scales linearly with the number of active servers: if 50% of the servers are active, the performance of Rabbit should be at least 50% of its maximum performance.

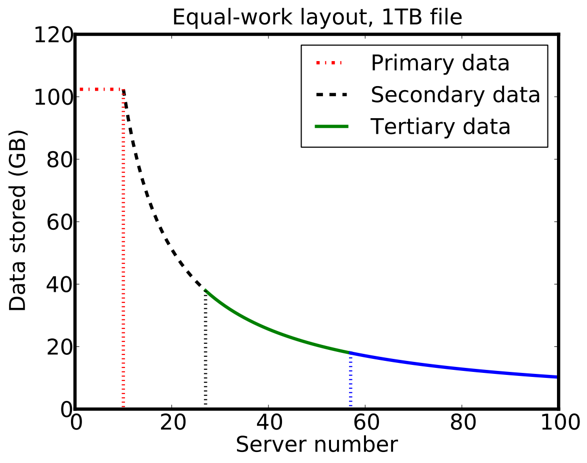

Figure 1: An example of the equal-work data layout storing 1 TB across 100 servers. Lower numbered servers store more data because, in low power modes, they are the only active servers; thus they must do a greater share of the I/O work.

Rabbit achieves availability and power proportionality by using an *equal-work* data layout, depicted in Figure 1. In this data layout, the first replica of every block is written to one of the *primary* servers, and the other replicas are placed on the non-primary servers, according to the distribution shown. By having a replica of every block on the primary servers, Rabbit is able to extract all servers except the primaries while maintaining data availability.

Additionally, the non-primary replicas are not assigned randomly across the non-primary servers. Instead, after the first replicas are assigned to the first p servers, the second replicas are assigned to the next ep servers, the third replicas are assigned to the next e^2p servers, and so on (if necessary). Thus, for example, extracting only the highest numbered servers tends to leave on-line more than one replica of all data.

To achieve power proportionality, Rabbit must ensure 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 x active servers, each server must store at least $\frac{B}{x}$ blocks so that reads can be distributed equally. This ensures (probabilistically) that when a large quantity of data is read, no server must read more than the others and become a bottleneck. By default, Rabbit always extracts and reintegrates servers in descending and ascending order, respectively. Thus, if server numbered x is active, so are all servers $1...x$. This leads to a data layout where a server numbered x stores at least B/x blocks, with the exception of the primary servers. Since a copy of all blocks must be stored on the p primary servers, they each store $\frac{B}{p}$ blocks.

While this layout allows all servers to perform the same amount of *read* work, it can create bottlenecks when writing new data. This is because lower numbered servers must store more data, and therefore must write more data as it is created. In an ordinary HDFS-like file system, every server (out of N) stores approximately the same number of blocks: $3B/N$ (assuming a replication factor of 3). Correspondingly, when the file system receives a W-block write request, each server writes $3W/N$ blocks, assuming ideal load balancing. In Rabbit, the p primary servers would write W/p blocks for the same W-block request. If $p < \frac{N}{3}$, as intended, then the primary servers must write more data than others and can become the bottleneck during writes. Everest-style write offloading provides the solution to this problem. Borrowing directly from Everest's scheme, Rabbit offloads writes temporarily from over-busy primaries to any active non-primary servers; with all servers active, maximum write performance is provided by having every server handle an equal fraction of block writes, and every server will have some data offloaded from primaries.

The Rabbit design addresses a number of other details, most of which are not crucial to the agility issues explored in this paper. For example, it includes I/O scheduling mechanisms for explicit load balancing of reads, since HDFS's implicit balancing based on block distribution does not work for Rabbit's unbalanced layout. It includes data layout refinements that minimize the number of additional servers that must be re-integrated if a primary server fails. It also accommodates multi-volume data layouts in which independent volumes use distinct servers as primaries in order to allow small values of p yet not limit storage capacity utilization to $3p/N$.

2.3 Agility issues for elasticity

Rabbit makes extensive use of write offloading, as discussed above. For clarity, we distinguish between two forms of write offloading, based on why they are used, which we have dubbed *availability offloading* and *performance offloading*. Availability offloading is used when the target server is currently not active. In this case, the offloading system will have to find another active server to store the replica being written. Performance offloading is used when the primary servers would otherwise be a bottleneck for writes, which can occur during bursts of write activity because there are few primary servers and a copy of every block must be written to a primary. Both forms of offloading affect agility.

Performance offloading and server extraction time: With performance offloading, some fraction of data (called the *offloading factor*) that would be written to the primaries (and possibly other low-numbered servers) is instead written in a load-balanced way across all available servers. After writing new data, the data layout will no longer match the equal-work layout. A data migration agent works in the background to restore the desired data layout. By adjusting the fraction of data that is offloaded vs. written according to the equal-work layout, Rabbit can trade off write performance for cleanup work.

Until the data migration agent restores the equal-work layout, the two properties that provide agility may be violated: there may be blocks with no primary replica, and there may be servers storing fewer than $\frac{B}{x}$ blocks. Until all blocks have a replica on a server numbered less than x, Rabbit cannot reduce the number of active servers lower than x—thus, performance offloading of writes has a direct impact on the work required before given servers can be extracted. Even if there is a replica of every block on a server numbered less than x, the system will not be power proportional until all these servers are storing at least $\frac{B}{x}$ blocks. Section 3 describes new read and write offloading policies that reduce the amount of cleanup work that must be done before these properties are restored, thereby increasing agility.

Effective server reintegration time: Restarting a server's software can be quick, but there is more to reintegration than that. 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 equal-work layout is part of full reintegration.

Our assumption is that servers that are extracted are usually intended to retain their stored content while away. This is certainly true of servers that are powered down when extracted. It is also true if the alternate use to which the server is put does not require using the same storage capacity, such as if it is the other resources (e.g., CPU and unassigned storage) being reassigned to other activities. We assume that some form of isolation (e.g., virtual machines or Linux containers [4]) will ensure that existing content is not corrupted by the alternate use, but some form of pre-reintegration data verification could be used as well.

Even assuming that all stored data remains on the reintegrated server, it is not all useful for serving reads. While the server was not active, any writes that modified its stored data was offloaded to other servers, via write availability offloading, leaving the reintegrated server's version invalid. Until new writes and/or background migration restores a full complement of blocks onto the reintegrated server, it may be unable to service a full share of the read workload. Section 3 describes policies for quickly making the reintegrated server a full contributor to overall performance.

3 JackRabbit policies

This section describes JackRabbit's policies designed to improve its elasticity and agility. First, it describes JackRabbit's use of read offloading and how much it reduces the need for performance offloading of writes. Second, it describes policies for performance offloading of writes. Third, it describes policies for availability offloading of writes.

3.1 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 performance targets. When applications simultaneously read and write data, JackRabbit can coordinate the read and write requests so that reads are preferentially sent to higher numbered servers that naturally handle fewer write requests. By taking read work away from the low numbered servers (which are the bottleneck for writes), JackRabbit can increase write throughput without changing the offloading factor. We call this technique *read offloading*.

The amount of read offloading that JackRabbit is able to perform depends on the number of servers we are offloading from, termed the *offloaded set*, and the number of servers we are offloading to, termed the *offloading targets*. Additionally, it depends on the amount of over-replication; that is, the number of blocks beyond $\frac{B}{x}$ being stored by each server not in the offloaded set. The analysis below assumes that there is no over-replication; each server is storing exactly $\frac{B}{x}$ blocks. Over-replication will only serve to increase the possibility of read offloading.

To determine the amount of read offloading that is possible, we consider the probability that a particular block can be read from a server outside of the offloaded set. Assuming a replication factor of 3, there are two possibilities to consider. The first is that the offloading targets are comprised entirely of either secondary or tertiary servers, but not a mix of the two. In this case, there is no duplication of blocks in the offloading targets, so all blocks stored on the offloading targets contribute to the amount of read offloading that is possible.

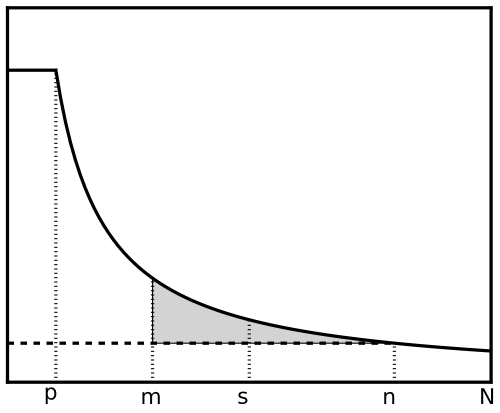

Figure 2: Offloading reads to servers between m and n, with servers n+1 through N inactive. The horizontal dashed line represents the number of blocks read from each active server without read offloading. The gray shaded area represents the amount of reads that can potentially be offloaded. s is the division between secondary servers and tertiary servers.

The second case occurs when the offloading targets are a mix of secondary and tertiary servers. This is depicted in Figure 2. In this case, it is possible that some blocks are replicated twice among the offloading targets, meaning that the number of blocks stored on the targets could be a poor estimate of the potential read offloading. Rather than computing how many blocks are being double counted, it is simpler to compute the probability that a block does not exist in the offloading targets.

Suppose the offloading targets are made up of the machines ranging from m to n (inclusive). We can divide these into two groups, the ones that are secondary servers [m, s], and the ones that are tertiary servers [s+1, n]. We can then compute the probability that a block's secondary replica is stored outside of [m, s] and its tertiary replica is stored outside of [s+1,n]. The probability that it is not in the set of offloading targets is the product of these two probabilities:

$$\left(\frac{\sum_{i=p+1}^{m} \frac{1}{i}}{\sum_{i=p+1}^{s} \frac{1}{i}}\right)\left(\frac{\sum_{i=n+1}^{N} \frac{1}{i}}{\sum_{i=s+1}^{N} \frac{1}{i}}\right)$$

3.2 Performance offloading of writes

Performance offloading is used to reduce the load on active, but overly busy servers. When using the equal-work data layout, the servers that host more data receive more write requests and can become a bottleneck for write performance. For instance, consider a cluster of 100 servers with 10 primary servers, where each server's storage system can write 90 MB/s. With 3 replicas of each data block, we would expect the maximum write throughput of user data in the system to be $\frac{(100)90\text{MB}/s}{3} = 3000\text{MB}/s$. However, in the base JackRabbit layout, 1 copy of each data block must be written to a primary server, so the write performance is actually $\frac{(10)90\text{MB}/s}{1} = 900\text{MB}/s$, less than a third of what it would be with a balanced data layout.

Performance offloading is a technique used when the target server is active, but is deemed to be a performance bottleneck. Rather than being sent to the server chosen by the equal-work layout agent, offloaded writes are sent to a server with low load, as determined by a load balancing agent. The algorithm for choosing when to offload a write, and where to send it, is know as the *offloading policy*. JackRabbit's write offloading policy has two key features. First, it bounds the set of servers for which any cleanup is needed before extraction to as small a set as possible, given a target maximum write throughput. Second, it provides peak write throughput controlled by a parameter called the *offloading factor*, for which the value can be dynamically tuned to trade-off between maximum write throughput and cleanup work. With a low offloading factor value, few writes will be offloaded, and little cleanup work will be subsequently required, but the maximum write performance will be limited. Conversely, a higher offloading factor will offload more writes, enabling higher maximum write performance at the cost of more cleanup work to be done later. An offloading factor of 0 writes all data according to the equal-work layout (no cleanup), and an offloading factor of 1 load balances all writes (maximum performance).

JackRabbit's offloading policy is motivated by two observations. First, since only a subset of servers are a bottleneck, we should only offload writes that would otherwise have gone to that subset of servers; writes going to non-bottleneck do not need to be affected by offloading. Second, only servers that hold the primary copy of data must be kept activated; selecting the servers to which particular offloaded writes are sent can avoid the need for any pre-extraction cleanup work for most servers.

In JackRabbit's offloading policy, the offloading factor indicates what fraction of the total number of machines should be in the *offload set*. This is the set of servers that receive the most write requests. Note that the offload set always includes the p primaries. When a write occurs, we first choose locations for the replicas according to the ordinary equal-work layout. We then modify the choice according to the following three rules:

1. The primary replica is load balanced across the entire offload set to ensure that all servers in the offload set handle the same rate of writes.[3]

2. Any non-primary writes to the offload set are instead load balanced among the remaining servers.

3. All remaining writes are unmodified; they are written to their originally chosen servers.

Figure 3 illustrates the write throughput that is offloaded, when the offload set consists of $m > p$ servers. The area marked P1 is offloaded from the p primary servers to servers $p+1$ to m (the area marked P1'), as per rule #1. All writes that would have gone to those servers (the areas marked P2) are offloaded to less-loaded higher-numbered servers (the area marked P2' + A'), as per rule #2. The A and A' refer to availability offloading, which is discussed in Section 3.3.

In addition to this policy, we have experimented with a similar, but slightly simpler policy. Like before, the simpler policy first chooses write locations based on the ordinary equal-work layout. However, instead of only writing the primary replica to the offload set, all writes that would have gone to the offload set are load-balanced across the offload set. Remaining writes are unaffected. While this policy is simpler, it writes more data to the offload set for any given offloading factor. Hence, it achieves lower maximum write throughput for the same amount of cleanup work (or, conversely, more cleanup work will be needed for any given maximum write throughput).

3.2.1 Modeling Write Performance

It is useful to be able to predict the performance that results from using a particular offloading policy and offloading factor. This can be used in designing new data layouts, predicting the performance at large scales, and choosing parameters to use for an existing system. We have observed that a simple rule can be used to model the overall write performance of the storage system: the write performance of the system is inversely proportional to the number of blocks written to the busiest server.

[3]The one exception is when the client requesting the write is in the offload set. In this case, JackRabbit writes the primary copy to that server to exploit locality.

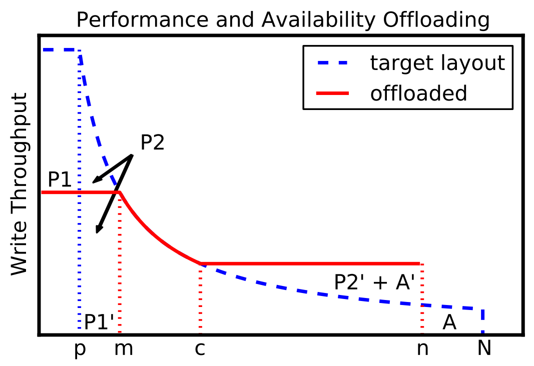

Figure 3: Write offloading for performance and availability. Writes for over-loaded primaries are load balanced across the offload set (servers 1 to m), and writes to non-primaries in the offload set (servers $p+1$ to m) are load-balanced over other servers. Writes for non-active servers are also load-balanced over other servers. Each such offloaded area is marked with a symbol and a symbol-prime, whose sizes must be equal (e.g., P1 is offloaded to P1').

Using this model, it is simple to calculate the maximum write throughput of JackRabbit. We know that the servers in the offload set are the bottleneck for writes. Furthermore, we know that exactly one replica of each block is written to this set of servers. Therefore, the number of blocks written to each server is $\frac{B}{fN}$, where B is the total number of blocks written, f is the offload factor, and N is the total number of servers. The maximum write throughput of the system is therefore proportional to $\frac{fN}{B}$.

3.3 Availability offloading of writes

When the equal-work layout policy tries to write a block to a host that is currently inactive, JackRabbit must still maintain the target replication factor for the block. To do this, another host must be selected to receive the write. JackRabbit load balances availability offloaded writes together with the other writes to the system. As illustrated in Figure 3, where the writes to inactive servers (area A) are offloaded to high-numbered servers (area P2' + A'), this results in the availability offloaded writes going to the less-loaded active servers rather than adding to existing write bottlenecks on other servers.

The same mechanism is also used to handle cases where the replication factor is set higher than it needs to be. To use the equal-work data layout as described, there must be a precise relationship between the number of primary servers, the total number of servers, and the replication factor. Specifically:

$$\sum_{x=p+1}^{n} \frac{1}{x} = R - 1$$

$$\ln \frac{n}{p+1} \approx R - 1$$

$$p \approx \frac{n}{e^{R-1}} - 1$$

However, it is likely that an administrator would like to set the number of primary servers to a value that would not yield an integer replication factor. For instance, with a cluster of 100 servers and 10 primary servers, the ideal replication factor would be approximately 3.26. Rounding down to a replication factor of 3 would create a data layout that is not power proportional. While it is possible to replicate some blocks 3 times and other blocks 4 times so that the average replication factor is 3.26, an administrator might want to increase the replication factor for improved fault tolerance. Figure 4 depicts an over-replicated layout. In

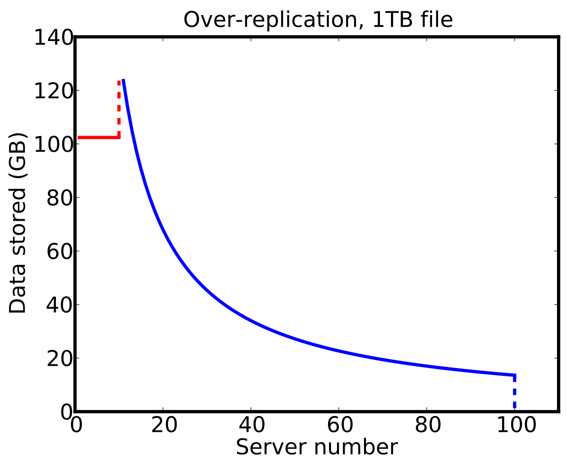

Figure 4: An over replicated layout without over-replication offloading. Because there are too many non-primary replicas, a naive placement strategy will create bottlenecks on the low-numbered non-primary servers.

these cases of over replication, the low-numbered non-primary servers receive more writes than the primary servers and become a bottleneck.

The availability offloading mechanism can be used to solve the performance problems caused by over-replication. This is done by telling the equal-work layout algorithm that there are more hosts than truly exist. If the layout agent assigns a block to a server that does not exist, it is treated as a write to an inactive server, and the availability offloading mechanism is used.

4 Implementation

JackRabbit is implemented as a modification of Rabbit, which itself is a modified instance of the Hadoop Distributed File System (HDFS), version 0.19.1. Rabbit implements and uses a *Scriptable Hadoop* interface that allows experimenters to implement policies in external programs that are called by Hadoop. This enables rapid prototyping of new policies for data placement, read load balancing, task scheduling, and re-balancing.

The primary Scriptable Hadoop on which we rely allows for an external programs to modify block placement decisions made by HDFS. Ordinarily, when an 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 HDFS namenode in Scriptable Hadoop is modified to start a configurable placement process known as the *replication script* when the namenode starts. The namenode communicates with this script 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 candidate datanodes to the placement script's `stdin`. The placement script 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 by the placement script.

For our experiments, JackRabbit's size is expanded or shrunk by setting an "activity state" for each datanode. On every read and write, the layout policies 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 JackRabbit 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.

JackRabbit's data layout policy, load balancer, resizing and migration agent are implemented as a python program that is called by the Scriptable Hadoop placement code. The layout script determines where to place blocks according to the equal-work layout and the offloading policy. The load balancer used in JackRabbit keeps an estimate of the load on each server by counting the number of requests that JackRabbit has sent to each server recently. Every time JackRabbit 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 multiplying them by a decay factor. While this does not give the exact load on each server, we find its estimates to be good enough for load balancing among homogeneous servers.

To facilitate data migration, we further modified HDFS to provide an interface to get and modify the current data layout. We export two metadata tables from the namenode, mapping file names to block lists and blocks to datanode lists, and load them into a SQLite database. Any changes to the metadata (e.g., creating a file, creating or or migrating a block) are then reflected in the database on the fly. When data migration is scheduled, the JackRabbit 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 do so, we modified the HDFS client utility to have a "relocate" operation that copies a block to a new server.

5 Evaluation

This section evaluates JackRabbit's offloading policies. Measurements of the JackRabbit implementation show that it can provide performance comparable to unmodified HDFS, that it's 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 JackRabbit's agility enables significantly reduced commitment of active servers for the highly dynamic demands commonly seen in practice.

5.1 Experiments with JackRabbit prototype

5.1.1 Experimental Setup

Our experiments were run on a Hadoop cluster of 31 machines. The 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, JackRabbit 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 JackRabbit 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, permuted 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 JackRabbit's performance to the unmodified HDFS setup with the same resources. Our experiments are

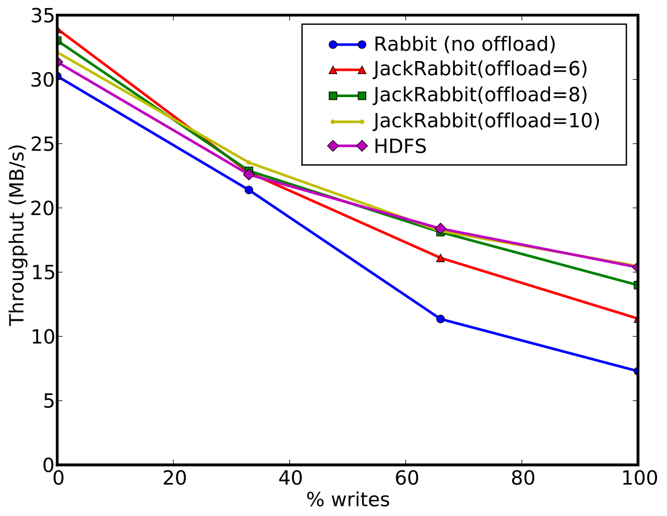

Figure 5: Performance comparison of Rabbit/JackRabbit with no offloading, original HDFS, and JackRabbit with varied offload setting.

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 agility-enhanced version can only be compared reasonably with it and not the capability of the raw storage devices.

5.1.2 Effect of Offloading policies

Our evaluation focuses on how JackRabbit's offloading policies affect performance and agility. We also measure the cleanup work created by offloading and demonstrate that its number of active servers can be adapted agilely to changes in workload intensity, allowing machines to be extracted and used for other activities.

Figure 5 presents the peak sustained I/O bandwidth measured for HDFS, Rabbit and JackRabbit at different offload settings. (Rabbit and JackRabbit are identical when no offloading is used.) In this experiment, the write ratio is varied to demonstrate different mixes of read and write requests. JackRabbit, Rabbit and HDFS achieve approximately the same 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 JackRabbit slightly outperforms the original HDFS due to read-offloading and explicit load tracking for balancing (given its uneven data distribution).

For Rabbit and JackRabbit with no offload, with the set of primaries constituting 13% of the cluster (4 of the 30 servers), the setup is highly elastic and is able to shrink 87% 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. JackRabbit provides the flexiblity to tradeoff some amount of agility for better throughput under periods of high load. As the write ratio increases, the effect of JackRabbit's offloading policies becomes more visible. Using only a small number of offload servers, JackRabbit 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 very agile—the cluster is still able to shrink 74%

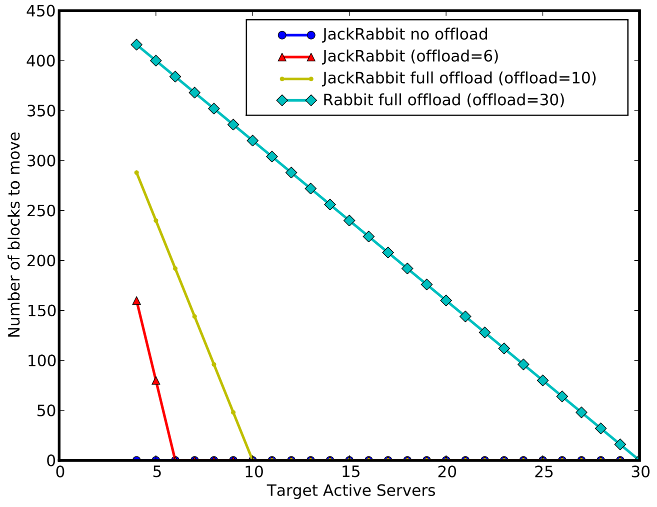

Figure 6: Cleanup work created by offloading. Each line shows the amount of cleanup work (in blocks) required to reduce the number of active servers from 30 to the X-axis value, for a different write offload setting. The line labelled "JackRabbit full offload (offload=10)" corresponds to JackRabbit-style offloading across the first 1/3 of the cluster to achieve even data distribution. The line labeled "Rabbit full offload(offload=30)" corresponds to the Rabbit offloading-to-all policy that offloads data to primary servers across the entire cluster. Zero blocks need to be moved when only non-offload servers are deactivated, and the amount of cleanup work is linear to the number of target active servers.

with no cleanup work.

Figure 6 shows the number of blocks that need to be relocated to preserve the availability of data when seeking to reduce the number of active servers, using JackRabbit's offloading policies. As desired, JackRabbit'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. JackRabbit configured 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 comparisonis Rabbit's full offload against JackRabbit's full offload. Both provide the cluster's full aggregate write bandwidth, but JackRabbit'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.

JackRabbit'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 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 JackRabbit read offloading policy, despite its simple nature, compares favorably and falls within 3% from optimal on average.

5.1.3 Agile resizing in JackRabbit

Figure 7 illustrates JackRabbit'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: one ("Ideal") where the full cluster stays active throughout the experiment and one ("JackRabbit") where the cluster size is changed with workload intensity. As expected, the performance is essentially the same for the two cases, with a small delay observed when JackRabbit reintegrates servers for the third phase. But, the number of machine hours used is very different, as JackRabbit 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 benchmarks writing 28 files, each of 2GB size (a total of 14GB written). The second benchmark is one SWIM job [11] randomly picked from a a series of SWIM jobs synthesized from a Facebook trace which reads 4.2GB and writes 8.4GB data. The third benchmark is also a TestDFSIO2 benchmark, but with a write ratio of 20%. The TestDFSIO2 benchmarks are I/O intensive but the SWIM job consumes only a small amount of the full I/O throughput. For the JackRabbit case, 4 servers are extracted after the first write-only TestDFSIO2 benchmark finishes (shrinking the active set to 3), and those servers are reintegrated after the SWIM job completes.

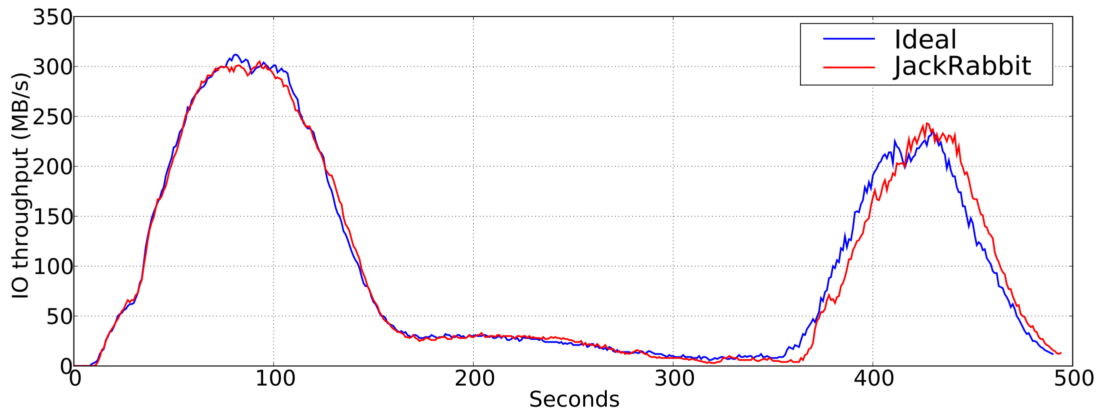

Figure 7: Agile resizing in a 3-phase workload

Figure 7's results are an average of 10 runs for both cases. The I/O throughput is calculated by summing read throughput and write throughput multiplied by the replication factor. Decreasing the number of active JackRabbit servers from 7 to 3 does not have an impact on its performance, since no cleanup work is needed. As expected, resizing 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 268 blocks are written to JackRabbit with only 3 active servers, but only 4 blocks need to be migrated to restore the equal-work layout. JackRabbit's offloading policies keep the cleanup work small, for both directions. As a result, JackRabbit extracts and reintegrates servers very quickly.

5.2 Policy analysis with real-world traces

This subsection evaluates JackRabbit 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

Table 1: Summary of traces. CC stands for "Cloudera Customer" and FB stands for "Facebook". HDFS bytes is calculated by 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

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 JackRabbit significantly reduces machine hour usage even compared to the state-of-the-art elastic storage systems, especially for write-intensive workloads.

5.2.1 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. [10]. Table 1 summarizes key statistics of the traces. The Facebook trace (FB) comes from Hadoop datanode logs, each record of which contains timestamp, operation type (HDFS_READ or HDFS_WRITE), and the number of bytes processed. From this information, we can calculate the aggregate HDFS read/write/total throughput as a function of time, where the total throughput is the sum of read throughput and write throughput mutliplied by the replication factor (3 for all the traces). The five Cloudera customer traces (CC-a through CC-b, using the terminology from [10]) 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. The Facebook trace was collected in 2010, and the five Cloudera traces in 2011.

5.2.2 Trace analysis and results

To simplify calculation, we make a few assumptions. First, the maximum measured total throughput in the traces corresponds to the aggregated maximum performance across all the machines in the cluster. Second, based on the first assumption, the maximum throughput 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 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 no cleanup work is needed, it proceeds to the next iteration; otherwise, it delays resizng until cleanup is done or performance requirement indicates increasing the active set, to allow additional bandwidth for necessary cleanup work. For increases in the active set of servers, it ensures that each newly reintegrated server has enough share of the data to fully contribute to handling read requests. It achieves this by increasing the cluster size ahead of time, so that newly reintegrated servers have enough time to fix the data layout and get ready to satisfy the higher throughput requirement.

Figure 10- 15 shows 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, JackRabbit, Rabbit and Sierra, respectively. We do not show the line for the Default HDFS, but since it is not elastic, its curve would be a flat line with the number of active servers always being the full cluster size (the highest value on the Y axis).

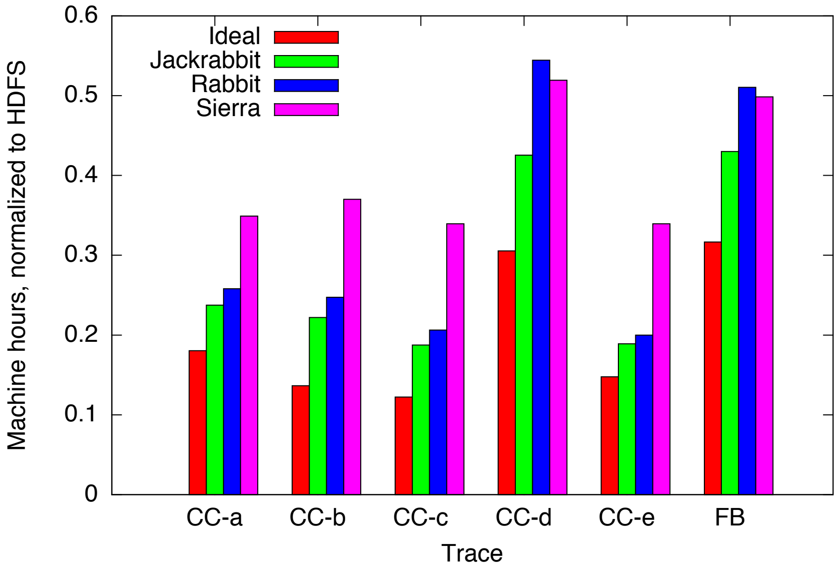

Figure 8: Total machine hours needed to execute each trace for each system. These results are normalized to HDFS, which must keep all servers active all the time.

As expected, JackRabbit exhibits much better agility than Rabbit, especially when shrinking the size of cluster, since it needs no cleanup work to resize to a count larger than the offload set. Such agility difference between JackRabbit and Rabbit is shown in Figure 10 at various points of time (e.g., minute 110 and 140) As a result, the JackRabbit line is always below or overlaps the Rabbit line, meaning that JackRabbit wins over Rabbit in terms of machine hour usage. The gap between the two lines indicates the number of machine hours saved due to the more agile data layout policies used in JackRabbit.

JackRabbit achieves lower machine hour usage than Sierra as well, as confirmed in all the analysis graphs— the JackRabbit line is always below or overlaps the Sierra line. 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. Nevertheless, JackRabbit can shrink the cluster size down to approximately 10% of the origianl footprint. When I/O activity is low, the difference in minimal cluster size can have a significant impact on the machine hour usage (e.g., as illustrated in Figure 13 and Figure 15). When expanding cluster size, Sierra incurs more cleaning overhead than JackRabbit, because deactivated servers are more under-stored than those in JackRabbitdue to Sierra's even data layout. These results are summarized in Figure 8 and Figure 9. Figure 8 shows the total number of machine hours used by each file system, normalized to the machine hours of HDFS. In each trace, JackRabbit outperforms the other systems, with the exception of "Ideal". Figure 9 shows the total amount of data migrated by Rabbit and JackRabbit while running each trace. JackRabbit always migrates less data than Rabbit. In most cases it reduced the amount of migration by a factor of 2 or more.

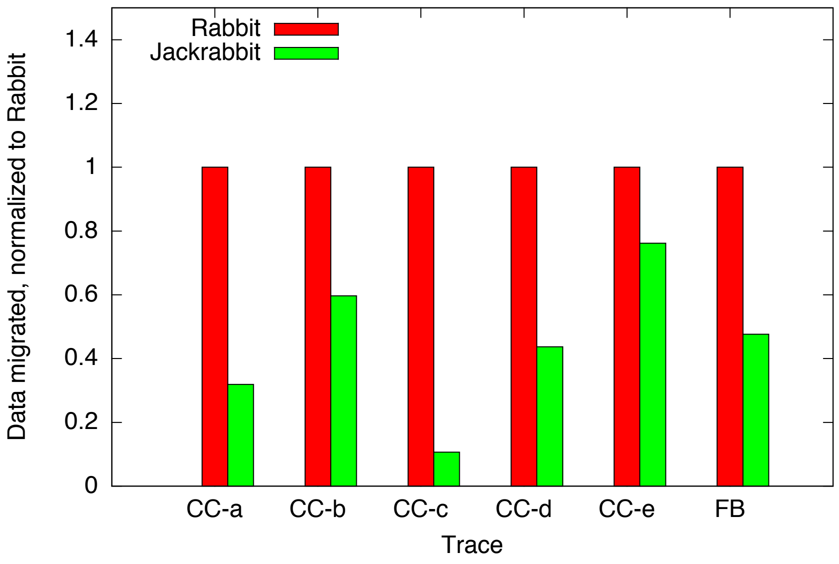

Figure 9: Total data migrated for Rabbit and JackRabbit, normalized to results for Rabbit. In all cases JackRabbit needs to migrate less data than Rabbit, often by a factor of 2 or more.

6 Conclusions

With appropriate offloading policies, elastic distributed storage can be agile: able to quickly extract and reintegrate servers. JackRabbit's offloading policies minimize the amount of data redistribution cleanup work needed for such resizing, greatly increasing agility relative to previous elastic storage designs. As a result, JackRabbit can provide the varying required performance levels in a range of real environments with much 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.

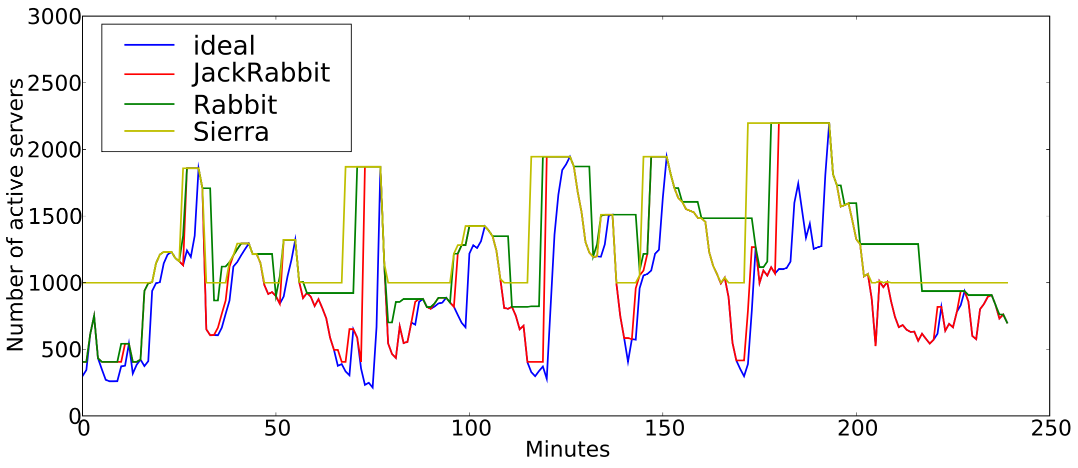

Figure 10: FB trace

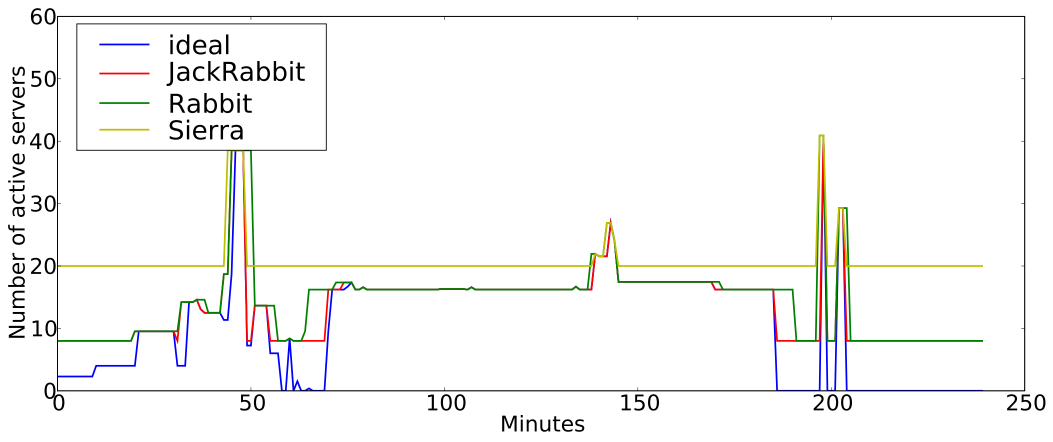

Figure 11: CC-a trace

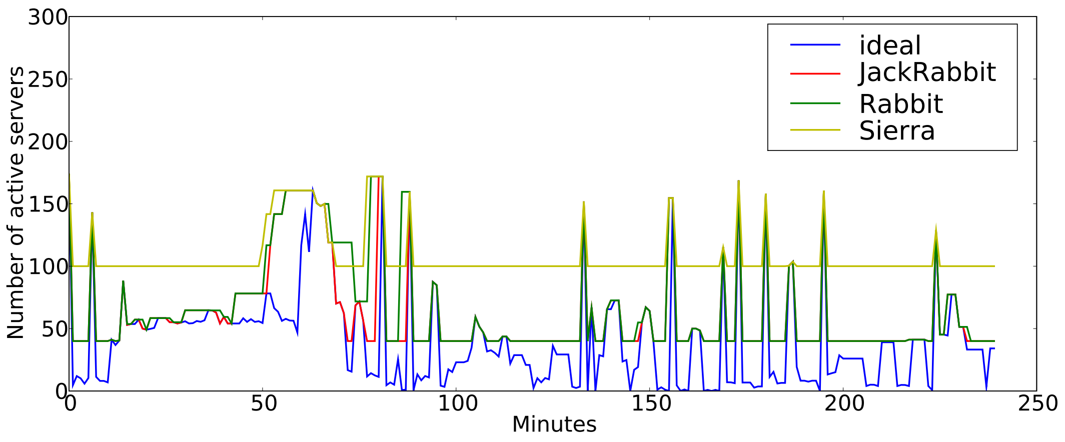

Figure 12: CC-b trace

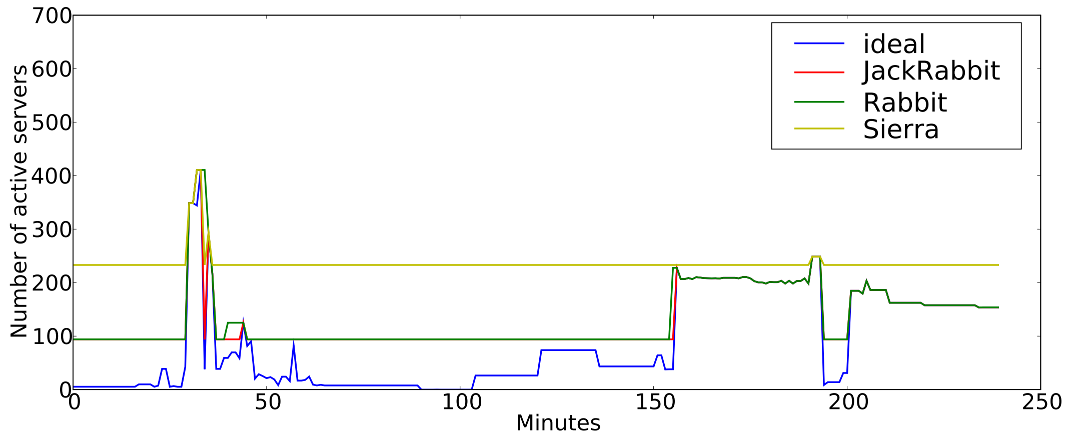

Figure 13: CC-c trace

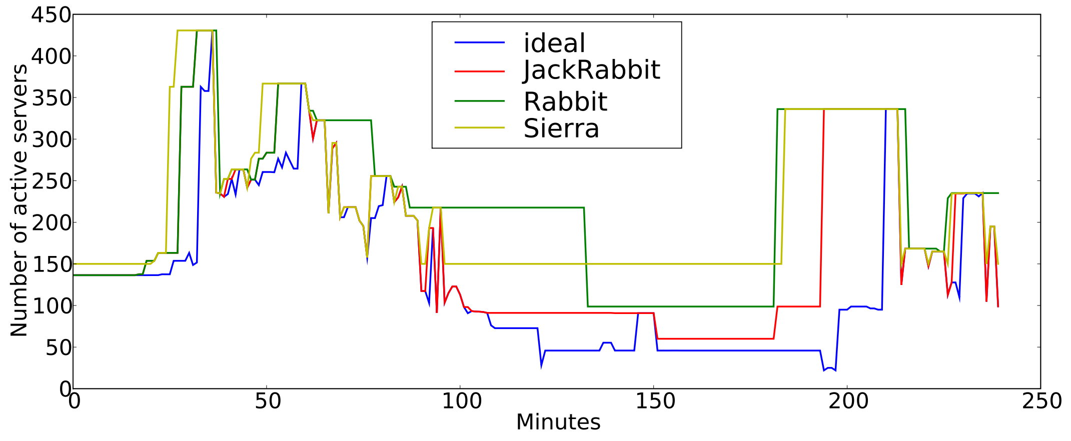

Figure 14: CC-d trace

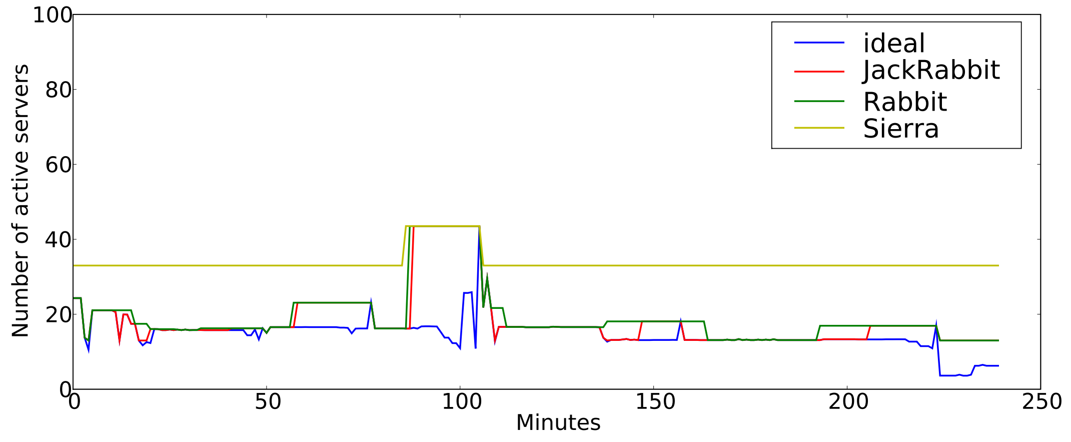

Figure 15: CC-e trace

References

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

[2] Amp lab, 2011. http://amplab.cs.berkeley.edu/.

[3] Istc-cc research, 2011. http://www.istc-cc.cmu.edu/.

[4] lxc Linux containers, 2012. http://lxc.sourceforge.net/.

[5] Mit, intel unveil new initiatives addressing 'big data', 2012. http://web.mit.edu/newsoffice/2012/big-data-csail-intel-center-0531.html.

[6] 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.

[7] D. Borthakur. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.

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

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. BigTable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):1–26, June 2008.

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

[11] Y. Chen and et al. The case for evaluating mapreduce performance using workload suites. *Mascots*, 2011.

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

[13] S. Ghemawat, H. Gobioff, and S. tak Leung. The google file system. *The 19th ACM Symposium on Operating Systems Principles*, pages 29–43, 2003.

[14] J. Leverich and C. Kozyrakis. On the Energy (In)efficiency of Hadoop Clusters. In *HotPower '09, Workshop on Power Aware Computing and Systems*, 2009.

[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 *USENIX Conference on Operating Systems Design and Implementation*, 2008.

[17] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: Building Distributed Enterprise Disk Arrays from Commodity Components. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 48–58, New York, NY, USA, 2004. ACM.

[18] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: A Power-Proportional, Distributed Storage System. Technical report, Microsoft Research, 2009.

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

[20] 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, NY, USA, 2009. ACM.

[21] 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*, 3(3), 2007.