Robust and Flexible Power-Proportional Storage

Hrishikesh Amur[†], James Cipar[⋆], Varun Gupta[⋆],
Gregory R. Ganger[⋆], Michael A. Kozuch[‡], Karsten Schwan[†]

[⋆]Carnegie Mellon University, [†]Georgia Institute of Technology, [‡]Intel Labs Pittsburgh

ABSTRACT

Power-proportional cluster-based storage is an important component of an overall cloud computing infrastructure. With it, substantial subsets of nodes in the storage cluster can be turned off to save power during periods of low utilization. Rabbit is a distributed file system that arranges its data-layout to provide ideal power-proportionality down to very low minimum number of powered-up nodes (enough to store a primary replica of available datasets). Rabbit addresses the node failure rates of large-scale clusters with data layouts that minimize the number of nodes that must be powered-up if a primary fails. Rabbit also allows different datasets to use different subsets of nodes as a building block for interference avoidance when the infrastructure is shared by multiple tenants. Experiments with a Rabbit prototype demonstrate its power-proportionality, and simulation experiments demonstrate its properties at scale.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management—*Distributed file systems*; D.4.5 [**Operating Systems**]: Reliability—*Fault tolerance*

General Terms

Design, Performance

Keywords

Cluster Computing, Power-proportionality, Data-layout

1. INTRODUCTION

Energy concerns have moved front-and-center for data centers. Much research and development is focused on reducing energy consumption, both by increasing efficiency and by using lower power modes (including "off") for underutilized resources. For the latter, an ideal is power-proportionality—the energy consumed should be proportional to the work completed. As cloud computing is used to aggregate many workloads on shared data center infrastructures, power proportionality is desired over a wide range of non-idle workload levels (i.e., system utilizations), down to a single operating tenant.

Our focus is on large-scale cluster-based storage (e.g., Google FS [9] or HDFS [1]) and data-intensive computing frameworks that are increasingly built on and co-mingled with such storage. Traditionally, such storage randomly places replicas of each block on a number (e.g., three) of the nodes comprising the storage system. This approach provides robust load balancing, fault tolerance, and scalability properties. But, it prevents powering-down of subsets of nodes—the primary tool for power proportional cluster-based storage—without disrupting data availability. Nearly all nodes must be kept powered-up.

This paper describes a power-proportional distributed file system (PPDFS), called Rabbit, that uses new cluster-based storage data layout to provide three key properties. First, Rabbit provides a wide range of power-performance settings, from a low minimum power to a high maximum performance, while maintaining the efficiency of a non-PPDFS design and fine gradation of settings. Second, for each setting, Rabbit provides ideal power-proportionality, meaning that the performance-to-power ratio at all performance levels is equivalent to that at the maximum performance level (i.e., all nodes powered-up). Third, Rabbit maintains near-ideal power proportionality in the face of node failures. In addition to its power-related properties, Rabbit provides a mechanism for mitigating contention among multiple workloads using distinct datasets.

Rabbit uses the *equal-work* data-layout policy, explained in detail in Section 3.1, which stores replicas of data on non-overlapping subsets of nodes. One replica, termed the primary replica, is stored on p nodes in the cluster, where p is arbitrarily small (but big enough to contain the replica). This allows the equal-work policy to have a low minimum power setting that only requires the p nodes to be kept on to guarantee availability of all data. The remaining replicas are stored on additional and increasingly-large subsets of nodes. This is done by fixing the order in which the nodes are turned on by creating an *expansion-chain* and storing a number of blocks on a node that is inversely related to where the node figures in the expansion-chain. This results in an even division of a workload across whatever subset of nodes is powered-on. Thus, performance can be scaled up in an ideally power-proportional manner, at the fine granularity of the throughput of one node at a time, to a maximum of roughly $e^{r-1}p$ nodes, where r is the number of replicas stored for that data. Scaling up performance is fast because it requires no data movement. Experiments show that Rabbit can provide ideal power-proportionality and full data availability across a range of settings going as low as 5% of the nodes if 4 replicas of the data are stored.

In very large-scale cluster-based storage, faults are viewed as a common occurrence rather than an uncommon one [8, 19]. Rabbit's equal-work data-layout policy carefully arranges secondary replicas of blocks to minimize disruption of power-proportionality when nodes fail. In particular, at the minimum-power setting, only the p nodes storing the primary replica are powered-up, and a failure of one of those p nodes necessarily requires turning on a number of secondaries to provide availability for the corresponding data. Rabbit's data-layout policy enables one to keep that number small and also addresses scalability of re-replication of the data that was stored on failed nodes. For example, by trading a small amount of flexibility regarding the granularity of power settings, one can keep the number below 10% of p.

Rabbit's mechanisms for providing power-proportionality can also be used for controlled sharing of I/O resources between applications. This is particularly useful in support of cloud computing, where there may be multiple, concurrent tenants that rely on the underlying distributed file system (DFS) for storage of independent datasets. For example, the DFS might be shared by various services in addition to a framework for execution of MapReduce [8], Hadoop [1], BigTable [6], or Pig [2] jobs. While such frameworks offer capabilities to schedule resources between their jobs, neither they nor the typical DFS provides mechanisms to allocate specific proportions of the DFS's I/O bandwidth among services and such frameworks (or even among jobs in a framework). A modified instance of Rabbit's equal-work data layout provides a building block for fair sharing or priority-based policies to be defined for access to different datasets. The capability to enforce fair-sharing of I/O resources between different datasets comes at the cost of a small loss of ideal power-proportionality, but experiments show that this cost is below 10% in the average case.

The three primary contributions of this paper are:

- The introduction and evaluation of the equal-work data-layout policy, and its realization in a PPDFS called Rabbit, capable of providing ideal power-proportionality for storage at arbitrarily low minimum-power settings.
- The introduction and evaluation of modified equal-work data-layout policies that enable recovery from disk/server failures in a power-proportional manner.
- The introduction and evaluation of modified equal-work data-layout policies that enable different applications to use non-overlapping subsets of nodes concurrently to avoid interference.

The remainder of this paper is organized as follows. Section 2 motivates our work in more detail. Section 3 describes the design of Rabbit and its data-layout policies. Section 4 evaluates their power-proportionality and other properties. Section 5 discusses additional related work.

2. NEED FOR NEW DATA LAYOUTS

The cluster-based storage systems commonly used in support of cloud and data-intensive computing environments, such as the Google File System(GFS) [9] or the Hadoop Distributed Filesystem [1], use data layouts that are not amenable to powering down nodes. The Hadoop Distributed File System(HDFS), for example, uses a replication and data-layout policy wherein the first replica is placed on the writer node (if it contributes to DFS storage), the second on a random node on the same rack as the writer, and the third on a random node in a remote rack. In addition to load balancing, this policy provides excellent availability properties—if the node with the primary replica fails, the second replica on the same rack maintains data availability; if an entire rack fails (e.g., through the failure of a communication link), data availability is maintained via the second or third replica, respectively. Unfortunately, this policy also prevents power proportionality by imposing a strong constraint on how many nodes can be turned off. In the case of default HDFS, no more than one node per rack can be turned off without the risk of making some data unavailable.

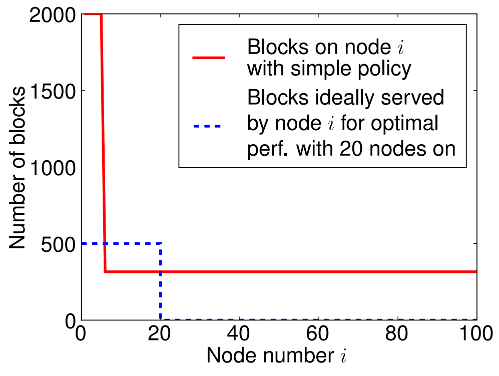
Alternate data layout policies are needed to accommodate power proportionality. This section discusses a basic power proportional data layout and issues involved with tolerating faults and managing I/O resource sharing. Section 3 builds on this background in describing the data-layout policies used in Rabbit.

2.1 A Simple PPDFS Data-layout Policy

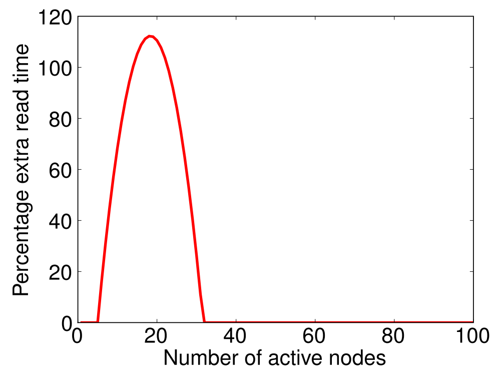
Suppose that we wish to store r replicas of B blocks of a dataset on the DFS that is running on a cluster of N nodes. We first try a simple and intuitive policy: we store one replica of B blocks evenly on p nodes, called the primary nodes, where $p \ll N$. The remaining $r-1$ replicas are distributed evenly over the remaining $N-p$ nodes. To show why this policy might work, consider an application that reads the entire dataset and therefore makes requests for each of the B blocks. If the cluster operates in the minimum power setting, then only p nodes will be kept on since turning off any more nodes makes some data unavailable. This corresponds to a low minimum power setting since $p \ll N$ and minimum performance since each node has to service B/p blocks. Additional nodes can be turned on for better performance. At the maximum performance setting, all N nodes in the cluster are on, and each of them services only B/N blocks. This "simple" PPDFS policy, used in some related work discussed in Section 5, is shown in Figure 1(a) for $N = 100, B = 10^4, p = 5$ and $r = 4$.

The key problem with this approach is that it is not ideally power-proportional, because there are cases where all the nodes that are 'on' cannot perform equal work. In Figure 1(a), with $n = 20$ nodes 'on' and the rest turned off, ideally each of the nodes would service $B/n = 500$ blocks. But each of nodes $i \in [10, 20]$ only stores about 316 blocks and of course, a node can service a request for a block only if it stores that block. Therefore, each of the primary nodes needs to service more than 500 blocks leading to an increase in the overall read completion time and hence a drop in overall throughput. The percentage increase in read time for such a policy is shown in Figure 1(b).

Therefore, while the goals of a low minimum-power, a high maximum-performance, and fast, fine-grained scaling are met, the policy described above is not ideally power-proportional. Ideal power-proportionality can be attained by storing the same number of blocks on a primary node as a non-primary node, which will allow all the nodes to



(a) Simple data layout arrangement



(b) Increase in read time for simple policy

Figure 1: Problems with a simple PPDFS data layout policy. For numbers of active nodes between p and approximately one-third of all nodes, power efficiency is low as the non-primaries contain too little data to contribute equally to providing service. Most requests must go to the primaries, resulting in significant load imbalance and performance reduction (relative to ideal) for reading the entire dataset.

perform an equal amount of work. However, doing so compromises on the goal of a low minimum power setting, because it necessitates $p = N/r$. But, it is preferable to keep the number of replicas r as low as possible. For example, a high value of r implies increased use of disk capacity and decreased write performance, since any changes have to be propagated to all the replicas. We show in Section 3.1 how the *equal-work* data-layout policy used with Rabbit achieves all of these desired properties for a PPDFS.

2.2 Power-Proportional Fault Tolerance

In addition to power-proportionality in failure-free periods, it is desirable for large-scale PPDFSs to provide power-proportional failure and recovery modes. As distributed file systems (DFS) are deployed across larger clusters, server failures become more and more common, and the amount of time the DFS spends in failure and recovery states increases. This has led users of some of the largest distributed file systems to comment that failure recovery states are now the norm, not the exception [8, 19]. To efficiently support such large file systems, a PPDFS must remain power proportional even when it is recovering from a failure. The simple policy fails in this regard: if a primary server fails, almost all of the non-primary nodes will have to be activated to restore availability of the data from that primary. This is because the data from each primary is spread evenly over all nodes, making it difficult to find a small set of nodes containing replicas of all of the blocks from the failed primary. In section 3.5, we explain this problem in more detail and describe modifications to the base Rabbit file system to allow it to restore availability with little effect on the power consumed by the system.

2.3 I/O Resource Management

The ability to allocate I/O bandwidth available through the DFS to *specific* applications that run on it would have significant benefits. Recent results [11] show that almost 5% of the jobs observed in a large-scale data center run for more than 5 hours and some jobs run for more than a day. In the presence of such long-running jobs, it is imperative to be able to guarantee some notion of fair sharing of the resources of the cluster. There should be capabilities, for example, to temporarily decrease the performance of long jobs during times of high load or when there are higher priority, shorter running jobs to be processed. Although Hadoop or an equivalent implementation of the map-reduce paradigm has its own scheduler, the underlying DFS will most likely support multiple kinds of applications in the data center. For example, Google's BigTable [6] and Hadoop's HBase are designed to work directly on top of the DFS. It is not possible, with current solutions, to guarantee I/O performance for each of these jobs. In other words, there is no check on a single job monopolizing the I/O resources of the cluster. This problem is often exacerbated by the fact that jobs are increasingly data-intensive, such that their overall performance depends significantly on the amount of I/O bandwidth that they receive.

The DFS is an ideal location for the implementation of mechanisms to control the amount of bandwidth provisioned to applications. Rabbit manages I/O resources between datasets stored in the cluster. It is possible to allocate I/O bandwidth to a particular dataset that would then be shared by the applications using that dataset. We describe the method in Section 3.4 and show benefits in Section 4.3.

3. DESIGN OF RABBIT

In addition to power proportionality, Rabbit, was designed to provide high bandwidth data I/O using commodity compute servers in a cluster environment, and to tolerate hardware failures. Consequently, Rabbit shares some properties with other recent cluster file systems, such as the filesystems of Google [9] and Hadoop [1]. In particular, files are divided into large blocks and a user-selectable number of replicas, r, of each data block is distributed among the nodes of the cluster. The typical modern cluster for which these file systems were designed may consist of thousands of servers, where each server stores data blocks on a small number of disks, generally less than 10. The mapping of file names to block identifiers is also maintained by a separate meta-data service.

Rabbit differs significantly from these other systems in providing power-proportionality. This property, which is especially attractive as cluster file systems are increasingly being employed for general cluster computing where significant power savings may be possible, induces changes to the data

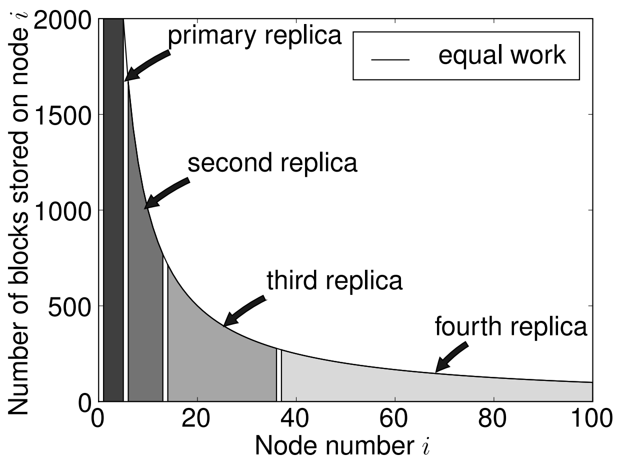


Figure 2: Equal-work data layout

layout policy, load balancing algorithms, and fault-tolerance techniques.

3.1 Equal-work Data Layout Policy

This section describes the *equal-work* data layout policy used in Rabbit. Consider a cluster with N nodes, where $tput_n$ is the I/O throughput obtained and pow_n the power consumed when n nodes are active (powered on). We state the requirements of a PPDFS formally:

1. A low minimum throughput, $tput_p$, consuming power pow_p, where p nodes are kept active and $p \ll N$.
2. A high maximum throughput, $tput_N$, consuming power pow_N, when N nodes are kept active.
3. Ideal power-proportionality, which means that $tput_i/pow_i = tput_N/pow_N$ for any $i \in \{p, ..., N\}$.
4. Fast, fine-grained scaling with no data movement required.

The *equal-work* policy, described next, ensures equal load-sharing. Formally, the equal-work policy is the result of an optimization problem that minimizes p with the constraints, $tput_i = (i/p)tput_p$ for all $i = p+1, ..., N$ for a given replication factor r. The following subsections offer an intuitive explanation of the equal-work policy. An example is shown in Figure 2 for the case of $N = 100, B = 10^4, p = 5$ and $r = 4$ as before.

3.1.1 Definitions

A *dataset* is an arbitrary user-defined set of files stored in the DFS. For each dataset, we define an ordered list of nodes, called the *expansion-chain*, which denotes the order in which nodes must be turned on or off to scale performance up or down, respectively. The nodes of the *expansion-chain* that are powered on are called the *active nodes*, $A(d)$, for dataset d. For the rest of Section 3.1, we do not consider multiple datasets, which will be discussed further in Section 3.4.

3.1.2 Low Minimum Power

In the equal-work data-layout policy, the first p nodes of the expansion-chain are called the primary nodes. One replica of the dataset, called the primary replica, is distributed evenly over the primary nodes as shown in Figure 2. Keeping only these p nodes on is sufficient for guaranteeing the availability of all data. Because $p \ll N$, this gives Rabbit a low minimum power setting.

3.1.3 Ideal Power-proportionality

To ensure ideal power-proportionality, $b_i = B/i$ blocks are stored on the i-th node of the expansion-chain, where $i > p$. This satisfies a necessary condition for ideal power-proportionality that is violated by the naïve policy, which is that b_i, the number of blocks stored by i-th node in the expansion-chain, must not be less than B/n for all $i \leq n$, when n nodes are active. Obeying this constraint makes it possible for the load to be shared equally among the nodes that are active. To illustrate, consider the situation when an entire dataset of B blocks has to be read from the DFS with $n \geq p$ nodes active. For ideal power-proportionality, each of the nodes should service B/n blocks. This is made possible by the *equal-work* layout because the n-th node stores B/n blocks, and each of the nodes i with $i \in [p, n)$ stores $B/i > B/n$ blocks. To scale performance up, the number of active nodes is increased by turning on nodes according to the order specified by the *expansion-chain* for the dataset. Scaling requires no data movement and can be done at the granularity of a single node.

3.1.4 High Maximum Performance Setting

Each node stores no more than the minimum required number of blocks, which allows the blocks to be distributed across a larger number of nodes while holding the number of replicas fixed so energy is not wasted writing an unnecessarily high number of data copies. We define a dataset's *spread* to be the number of nodes over which the blocks of that dataset are stored. A dataset's spread is equal to the length of its expansion-chain. For the equal-work policy, the spread depends on the number of replicas used.

We can derive a lower-bound on the spread based on the observation that the number of blocks stored on the servers in the range $[p+1, s]$ must correspond to $(r-1)$ replicas of the dataset. Hence,

$$\sum_{i=p+1}^{s} B/i = B(r-1) \tag{1}$$

Because $1/i$ is a monotonically-decreasing function, we also have convenient lower and upper bounds on $\sum_{i=p+1}^{s} 1/i$ as,

$$\int_{p+1}^{s+1} (1/x)\,dx \leq \sum_{i=p+1}^{s} 1/i \leq \int_{p}^{s} (1/x)\,dx \tag{2}$$

From Equations 1 and 2, we get:

$$s \geq pe^{r-1} \tag{3}$$

Note that the spread increases exponentially with the number of replicas while maintaining ideal power-proportionality. Since the maximum throughput obtainable depends on the spread, this allows the equal-work policy to obtain a high value for the same. We note that, since the spread also depends on p, a spread spanning the entire cluster can be obtained with any number of replicas r by adjusting the value of p.

3.2 Load Balancer

The equal-work layout policy is complemented by a load balancer, whose function is to ensure that each active node services almost the same number of blocks.

When a request is received from a client for a block, the DFS has a choice of which node will service the request, because each data block is stored on r different nodes. Since a typical data layout policy, like the default HDFS policy that is not power-proportional, stores data blocks evenly on all nodes, this decision can be made easily. In our case, some nodes store significantly more data than others, but the work should still be shared equally among all the active nodes. Therefore, when n nodes are active, although the n-th node has only $b_n = B/n$ blocks from a dataset with B blocks, it must service the requests for the same number of blocks as a primary node which stores B/p blocks. While a solution to an optimization problem formulated as a mixed integer program(MIP) can be used to optimally assign blocks to specific nodes for service, we find that the following heuristic works well in practice. We define a *desired hit-ratio*, equal to $\frac{B/n}{b_i}$ for each node $i \leq n$. When a dataset is being read, the *actual hit-ratio* at a given time t, denotes the ratio of the blocks serviced by node i to the number of blocks that could have been serviced by node i till time t. Of the r possible nodes that a requested block is stored on, some may be turned off so the choice is made from the nodes that remain on. The load-balancer greedily chooses the node for which the actual hit-ratio is the farthest from the desired hit-ratio. For example, if n nodes are active, the n-th node stores B/n blocks. Therefore, it has a *desired hit-ratio* of 1, which means any time the node appears as a candidate, it is very likely to be chosen. Similarly, a primary node has a *desired hit-ratio* of p/n. As one example evaluation of this approach, we performed complete reads of a 100GB dataset with 900 blocks on a cluster of 24 nodes, stored using the equal-work layout policy with our load balancer. For each run, we measured the maximum number of blocks serviced by any given node, as the overall throughput is inversely proportional to the maximum number of blocks serviced by a node. With our load balancer, the mean of the maximum number of blocks serviced by a node was 41.86 blocks compared to an ideal of 37.5 with a variance of 0.42.

3.3 Write Offloading

To this point, we have focused on read performance. However, a power-proportional DFS, and the design of the equal work layout, present two problems for write performance. First, a copy of every block must be written to a primary server. If the primary set is very small, as one would like it to be, then these nodes will become a bottleneck for writes. This becomes especially at high performance levels when most, if not all, of the nodes are on. In a system with 3-way replication, a perfectly power proportional DFS would get the write bandwidth of $\frac{1}{3}$ of the servers. However, if the primary set is small, for instance only 10% of the cluster, then the system will only achieve a write bandwidth of $\frac{1}{10}$ of the nodes.

The second problem occurs only when operating at reduced power modes. In these cases, it is impossible to write a block to an inactive node. If the layout policy requires that some data be placed on inactive nodes, the file system would have to activate them to do so. This layout requirement does not depend on the rate of write or read requests; there could be very little file system activity, but the layout still requires writing to a node that is currently inactive. Activating these servers will violate power proportionality, because the throughput achieved by the file system will remain quite low, while the power used by the system will increases.

We use a solution from other research to solve these problems: *write offloading* which writes to any available server, and corrects the layout later. Write offloading was used by Everest [15] to avoid bottlenecks caused by overloaded storage volumes in an enterprise storage setting. The problem of overloaded storage volumes is conceptually similar to our first problem (i.e. overloaded primary servers). Write offloading solves the bottleneck problem by allowing the primary replica of a block to be temporarily written to a non-primary server. Before that non-primary server can be deactivated, all primary copies of blocks must be written to their appropriate places. At first glance this is not solving the problem of overloaded primaries, but simply delaying it, since all blocks must reach a primary server eventually. However, the only reason the file system would deactivate a non-primary is when there is idle disk bandwidth. When this is the case, it can use that idle bandwidth to migrate the primary replica. The only disadvantage is that the file system will not be able to switch power modes instantly.

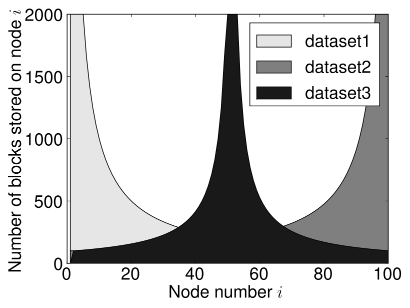
Write offloading also solves the second problem (i.e. writing to deactivated servers). This was demonstrated by PARAID [22] for writing to deactivated disks in a power aware RAID setting. It was also used by Sierra [20] in a setting very similar to our own: a large-scale distributed file system. The major challenge in these cases is that data written while in a low power mode will reside on a smaller-than-normal set of servers. This lowers the effective maximum performance for that data, until they can be migrated to the target layout. Thereska et al. [20] explore this in detail and show that this is not a problem for many real-world workloads.

3.4 I/O Scheduling

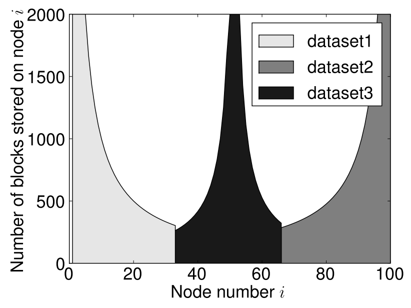
This section describes how we use the mechanisms used to provide power-proportionality to perform I/O scheduling for datasets in Rabbit. Recall that a dataset is defined to be an arbitrary set of files stored in Rabbit. We assume that all data entering the system is tagged with meta-data specifying the dataset that the data belongs to. One way to do this is to define datasets based on file system hierarchy, with subtrees explicitly associated with datasets, as with volumes in AFS [23].

Our focus is on data-intensive jobs whose performance significantly depends on I/O bandwidth, such as most jobs run on Hadoop. Hadoop has its own fair scheduler that indirectly manages I/O resources by controlling the compute scheduling, but this approach only guarantees fairness for map-reduce jobs using the particular instance of the Hadoop library. In a data center environment, there can exist multiple different applications, such as BigTable [6], that use the services offered by the DFS. In such scenarios, indirectly managing the I/O resources through compute scheduling becomes impossible. Our solution enables scheduling of I/O resources at the level of the DFS and allows the I/O bandwidth of the cluster to be shared among the datasets in an explicit manner.

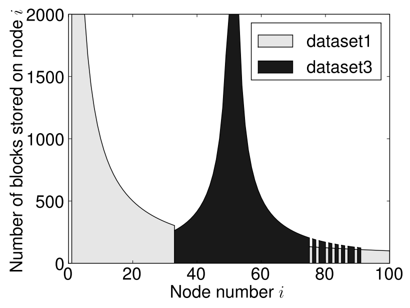
Section 3.1 explains Rabbit's equal-work data layout pol-



(a) Data layout for multiple datasets



(b) Fair sharing with 3 live datasets. Shaded regions represent the active nodes.



(c) Fair sharing with 2 live datasets. Shaded regions represent the active nodes.

Figure 3: Example multi-dataset layout involving three datasets

icy. To handle multiple datasets, we use the same policy but overlay the datasets over one another using a greedy strategy to choose the nodes. We define a *score*, s_i for a node i that depends on where that node figures in the expansion-chains of the different datasets. Let D_i be the set of datasets that have blocks stored on node i, let $s_i(d)$ be the contribution of dataset $d \in D_i$ to the score s_i, and let $l_i(d)$ be the index of node i in the expansion-chain of dataset d where node i appears. Then:

$$s_i = \sum_{d \in D_i} s_i(d) \tag{4}$$

$$s_i(d) = \begin{cases} 1 & \text{if } l_i(d) \leq p\,, \\ \frac{1}{l_i(d)} & \text{otherwise}\,. \end{cases} \tag{5}$$

When a new dataset is to be written into the DFS, nodes are chosen greedily starting with the node with the minimum score, s_i. The score's are updated once the blocks of the new dataset are stored. Figure 3(a) shows the layout policy for three datasets. Each of the datasets has a spread that is equal to the size of the entire cluster, but the order of nodes in the expansion-chains of the datasets is unique for each dataset.

To maintain control over the I/O bandwidth allocated to a dataset, a given node is assigned to *exactly one* dataset, which means that the I/O bandwidth of that node is allocated solely to that dataset. We choose this approach for two reasons. First, in a large cluster, the node will often be an acceptable level of granularity. Second, performance insulation in storage clusters remains an open problem, and sharing nodes relies on it to be solved.

We define a dataset to be *live* at a given time if an application is reading or writing data of that dataset. The set of active nodes, $A(d)$, is the set of nodes that have been allocated to dataset d and remain 'on'. The goal of I/O scheduling is, therefore, to allocate $A(d)$, for each of the datasets $d \in D_L$ where D_L is the set of live datasets. Since a node can only be allocated to one dataset, an arbitration algorithm is required if multiple, live datasets store blocks on a particular node i. We make this choice, with one exception, by picking the dataset d_0 where $s_i(d_0) = \max s_i(d)$, with $d \in D_i \cap D_L$. That is we pick the live dataset that contributes the most to the score of the node. Compensation scores, in proportion to $s_i(d)$, are added to to all datasets $d \in D_i \cap D_L$ that were not chosen. The exception to this rule is when the dataset d_0 has the least compensation score among the datasets in $D_i \cap D_L$, in which case the dataset with the maximum compensation score is chosen. For instance, if all three datasets shown in Figure 3(a) are live, fair-sharing would set the active nodes of the datasets as shown in Figure 3(b).

Rabbit controls the I/O bandwidth available to a dataset d by controlling the size of the set $A(d)$. Since all requests for blocks belonging to that dataset are serviced by the nodes in the set $A(d)$, and no others, this sets the total amount of bandwidth available to the dataset.

A side-effect of providing resource guarantees to datasets is that ideal power-proportionality does not hold in all cases. Consider, for example, if datasets 1 and 3 from Figure 3(a) are live. In this case, each of these should get the I/O bandwidth of 50 nodes. But, the first 50 nodes from $A(1)$ and $A(3)$ have many nodes in common. Since each node can only be allocated to one live dataset, each of the datasets has to settle for some nodes that are lower in their respective expansion-chains. The resulting sets of allocated active nodes are shown in Figure 3(c). The active nodes for dataset 1 that are lower in the expansion-chain cannot perform an equal share of work to that of the nodes higher in the chain. For instance, in Figure 3(c), in a read of the entire dataset each of the 50 active nodes is expected to service $B/50$ blocks. But the active nodes of dataset A that fall below the 50th node in the expansion chain have fewer than $B/50$ blocks, causing the higher-numbered nodes to perform extra work. This leads to a loss of ideal power-proportionality. We quantify this loss in power-proportionality by defining a *degree of power-proportionality.*

Consider a dataset of B blocks. Let n nodes from the expansion-chain be active. Ideally, each of the n nodes would service B/n blocks. But due to the overlapping of multiple datasets, a situation such as the one pictured in Figure 3(c) may result in the n-th node not storing the requisite B/n blocks. In this case, the "extra work" consisting of blocks that cannot be serviced by nodes lower down in the expansion-chain has to be distributed among the nodes with more blocks stored. If b_e is the number of extra blocks that each of the these nodes has to service, then the degree of power-proportionality is defined as $\frac{B/n}{B/n+b_e}$.

To understand the extent of this loss in power proportionality, we built a data layout simulator that allows us to understand the factors on which the degree of power proportionality depends. In particular, we investigate the dependence on the total number of datasets stored in the cluster and the number of live datasets. We show in Section 4.3.1 that the loss in efficiency is not more than an average of 10%.

3.5 Fault Tolerance

This section we will describes modifications to the equal work layout that allow the file system to remain power proportional when a primary server fails. We will only be considering crash failures, and not arbitrary Byzantine failures. The failure recovery process is composed of three parts, though they are not necessarily separate activities. Each involves restoring some property of the file system:

- **Availability:** all data may be accessed immediately. In the case of a PPDFS this means ensuring that every block is replicated on at least one active node.
- **Durability:** The file system's fault tolerance configuration is met. For Rabbit, this means that each block is replicated r times.
- **Layout:** The file system's target layout is achieved. For the equal-work layout policy, non-primary node i has approximately B/i blocks on it.

Most of this discussion focuses on availability. Restoring durability and layout after a primary failure uses the same mechanisms as writing new data to the file system, described in Section 3.3.

As it has been described, the equal-work data layout cannot remain power proportional in the event that a primary server fails, because blocks from each primary server are scattered across all secondary servers. When a primary server fails, all secondary servers must be activated to restore availability. Therefore, the non-fault-tolerant version of the equal-work layout cannot achieve its target minimum power setting of p when there is a primary server failure. Instead, it has a minimum power setting of $ep - 1$.

To avoid this outcome, we impose further constraints on the secondary replicas of each block. The secondary servers are grouped into *gear groups* and *recovery groups*, with each server belonging to exactly 1 gear group and 1 recovery group. To visualize this, imagine arranging the secondary servers in a grid configuration depicted in Figure 4. The rows of this rectangle are the gear groups, and the columns are the recovery groups. The number of servers in these groups, i.e. length of the rows and columns of the grid, are respectively known as the *gear group size* and *recovery group size.*

Each primary server is mapped to exactly 1 recovery group. All data hosted by that primary server is replicated across its corresponding recovery group. This leads to a simple failure recovery strategy: if a primary server fails, the file system activates the corresponding recovery group. Because all of the server's secondary replicas reside in that recovery group, this is sufficient to restore availability to the data stored on the failed primary.

In this layout scheme, gear groups are the basic unit of power scaling. It is not helpful to turn on extra replicas for some primary server's data and not others: work can never be shared equally if some primaries have to read all of their blocks and others have to read only some of their blocks. Therefore, when turning on servers to increase the power

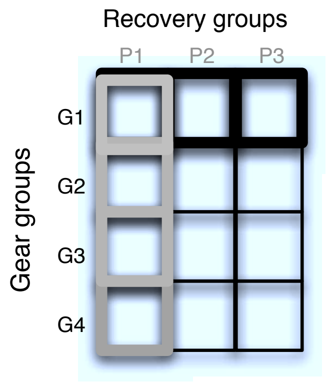


Figure 4: Gear groups and recovery groups. All data from a single primary exists on a single recovery group, such as the grey box. When increasing the power setting, the file system turns on gear groups in an all-or-nothing fashion.

mode, the file system must turn on a set of servers that will contain data from all primaries, i.e. a gear group.

To share work equally each gear group should contain approximately the same amount of data from each primary. The amount of data stored on each server in a gear group depends on where that gear group falls in the expansion chain. Servers belonging to low-numbered gear groups must store more data than those in high numbered gear groups, because they may be activated at lower power modes. If the last server in a gear group is server number i, then every server in the gear group stores $\frac{B}{i}$ blocks. Equivalently, with a group size of g, each server in gear group j stores $\frac{B}{(p+gj)}$ blocks.

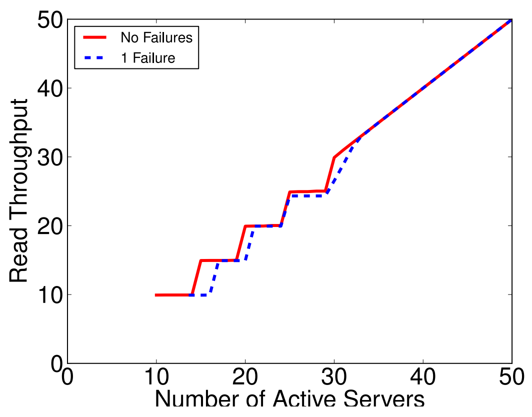


Figure 5: Simulated performance of fault-tolerant layout with 10 primary servers and a gear size of 5. The geared layout achieves very nearly ideal performance when a full gear is turned on, but less than ideal when a gear is only partially enabled.

Figure 5 shows the results of a simulation of the fault tolerant layout in a failure free case and with a single primary server failure. The performance is measured relative to the performance of a single server. The steps in the solid line show the effect of gearing: increasing the power setting

Gear size	Recovery group size
1	174
5	37
10	20
20	11
50	6
100	4

Table 1: Some example gear sizes and the corresponding size of the recovery group. This example assumes that the file system is configured with 100 primary servers. Even with a gear size of 5, allowing very fine grained scaling, the difference in minimum power setting is only 37%.

causes no improvement in performance until a gear is completely activated, at which point the performance jumps up to the next level. The dotted line represents the power and performance curve in the case of a single primary failure. The file system can achieve the same performance with only a moderate increase in power.

This data layout creates a trade off between gear size and recovery group size. A smaller gear size implies a larger recovery group size. By setting the gear size very small, we can achieve the goal of fine grained power settings, but the large recovery group size means that in the event of a failure the minimum power setting will be high. On the other hand, a large gear size does not allow fine-grained power adjustments, but can run at very low power even when recovering from a failure. This relationship is complicated by the fact that the number of secondary servers depends on the gear size. Recall that the amount of data on each secondary server in gear j is $\frac{B}{(p+gj)}$, meaning that choosing a larger gear group size causes less data to be stored on each server, thus requiring more secondary servers overall.

Table 1 shows the relationship between the gear size and recovery group for an example file system with 100 primary servers. The size of the recovery group, as a function of the number of primaries p and the gear size g, is $\frac{e(p-g)-p}{g}$. As an example from this table, if the gear size is 10% of the number of primary servers, the recovery group size will be about 20% of the primary size. This means that the minimum power setting during failure recovery is only 20% higher than the minimum power setting with no failures. The ideal setting of these parameters depends on the workload of the file system and the rate of failure, but these results show that there is a wide range of reasonable settings for these parameters.

4. EVALUATION

This section evaluates the different contributions made in this paper, including the power-proportionality of the equal-work data-layout policy, the improvements due to the introduction of gearing for fault tolerance and the feasibility and benefits of the added capabilities to perform I/O resource management at the DFS level. Our testbed for the experiments consists of a rack of 25 servers, each consisting of dual Intel Xeon E5430 processors and 16GB of RAM. For the experiments, we use two SATA disks connected to each node. The interconnect is a Force10 S50 Gigabit Ethernet switch. For the experiments on power-proportionality, the inactive servers are hibernated. For our servers, the power consumed in the hibernated state is approximately 5W and

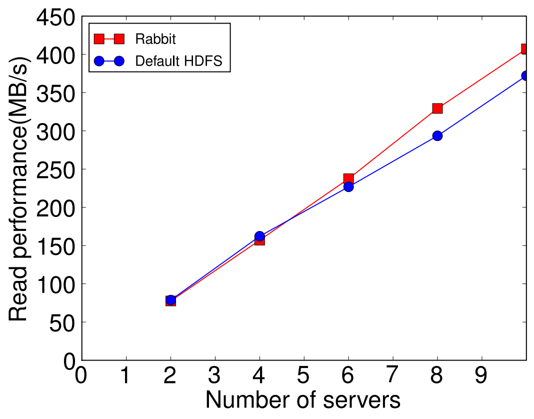


Figure 6: Read only performance of Rabbit at various power settings. HDFS cannot scale dynamically, and had to be re-configured before each run, while the power setting of rabbit can be set without restarting the file system.

time to wake up from hibernate is around 40s. Since servers are not usually expected to be hibernated often, the latency represents an unoptimized value. As hibernation of servers becomes more popular, we expect the wake-up times to decrease.

4.1 Implementation

We have implemented a prototype PPDFS called Rabbit. Rabbit is based on the Hadoop DFS [1], with modifications to the layout and load balancing. We converted the existing Java class for choosing block locations into an interface with which different data layout policies can be used. We implemented the equal-work policy and its corresponding load balancer so as to minimize the role of the meta-data server. The concept of datasets, which are arbitrary sets of user-defined files, was added to HDFS. Rabbit allows the number of active nodes for each dataset to be specified from the command line, so that an I/O scheduling policy can easily be written in user-space. We have not yet implemented the *write offloading* policy that is used to improve the performance of writes, but expect that it would behave as reported by Thereska et al. [20].

4.2 Power-proportional Operation

To evaluate the performance of the equal-work data-layout policy, we first test the peak I/O bandwidth available from Rabbit and compare it with that available from default HDFS on the same hardware, using a microbenchmark that uses the HDFS shell to perform I/O in a distributed manner. We then run a larger scale experiment using the Hadoop Terasort benchmark to evaluate power-proportionality. For Rabbit, the dataset is written into the DFS and the number of active nodes is set using a command-line utility. The remaining nodes can be hibernated to save power. For default HDFS, the data is written onto a number of nodes equal to the number of active nodes in the Rabbit case being compared with. The Linux buffer cache is cleared between runs.

4.2.1 *Microbenchmarks*

We test read performance on a 40GB sized dataset with a replication factor of 3. Figure 6 shows that Rabbit offers read performance equal to default HDFS while offering the benefit of power proportionality. The write performance of Rabbit is, however, not equal to the default case, owing to the imbalance in the amount of data written to the nodes. As discussed in Section 3.3, techniques developed for Everest [15] can be used to temporarily offload writes from primaries to the non-primaries and transfer the data lazily during times of low system load.

4.2.2 *Terasort*

We also evaluate Rabbit with the Hadoop Terasort benchmark, which, given the size of our testbed, adjusted to sort 100GB. For default HDFS, as before, we reload the dataset for each run, since HDFS cannot be dynamically scaled. We perform each run 3 times and report the median value. Figure 7 shows the results. The default HDFS sort performance is better than Rabbit due to the large write involved when the sorted dataset is written back to the DFS. For comparison, we also include times for the map phase of the computation both cases. The map phase involves reading the input files and generating intermediate data that is typically written by the node locally. Figure 7 shows that the times for the map phase in the two cases are comparable.

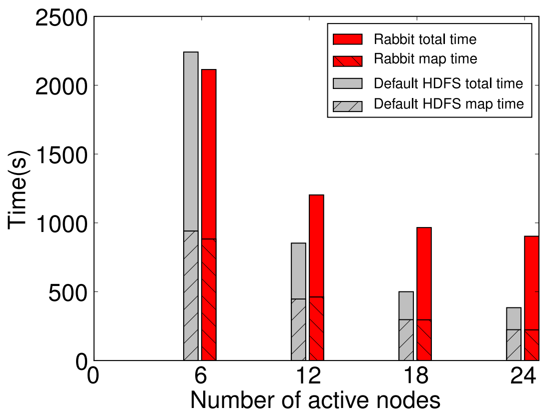


Figure 7: Time to sort a 100GB file using Hadoop. Map time (hatched) depends only on read performance, and is comparable between Rabbit and HDFS. Reduce time involves writing results back to the file system. Since we have not yet implemented write offloading in our prototype, the reduce phase is much slower on Rabbit.

4.3 I/O Scheduling

This section demonstrates the benefits of the capability to schedule I/O resources explicitly between datasets. By providing the mechanisms to do so, Rabbit provides the freedom to use an arbitrary scheduling policy to allocate resources. We implemented and evaluated two simple policies: fair-sharing and priority- scheduling.

4.3.1 *Factors affecting power-proportionality*

The ability to schedule I/O resources to specific datasets is gained at the cost of ideal power-proportionality. The desirability of the mechanism, therefore, depends on the sensitivity of the degree of power-proportionality to factors such as the total number of datasets stored in the cluster and the number of live datasets. With this in mind, we built a cluster simulator that stores datasets according to the equal-work data layout policy. Any combination of the stored datasets may then be chosen to be live, after which the sets of active nodes for each of the live datasets are assigned according to the policy explained in Section 3.4. The degree of power-proportionality is then evaluated for each of the live datasets by simulating an entire read of the dataset and calculating the amount of extra work that its primaries must perform to compensate for the unequal sharing. For a given setting of the total number of datasets and the number of live datasets, the simulator tests all combinations of datasets and calculates the average and worst case values for the degree of power-proportionality.

To validate the simulator, we replicated a subset these experiments on a cluster running Rabbit. Figure 8(a) shows predicted and experimental values for the dependence on the total number of datasets. Figure 8(b) shows them for the number of live datasets for a 14-node cluster. The experimental values are each within 5% of the predicted values which validates our simulator. In both cases, the average degree of power-proportionality is more than 90%. With our simulator validated, we also show that the average value of the degree of power-proportionality remains above 90% for larger clusters. Figure 9(a) shows the predicted values.

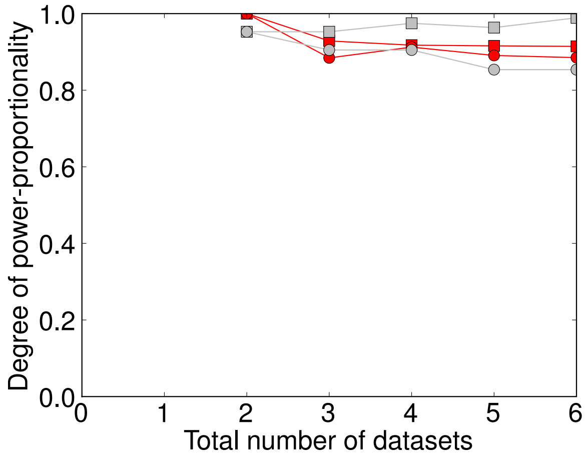
Having shown the feasibility of I/O resource management from the DFS, we evaluate two example I/O scheduling policies implemented on top of the allocation mechanism.

Instances of app_B	app_A Throughput(MB/s)	
	Rabbit	Default HDFS
1	180.3	137
2	175.3	101.2
3	173.6	82.5

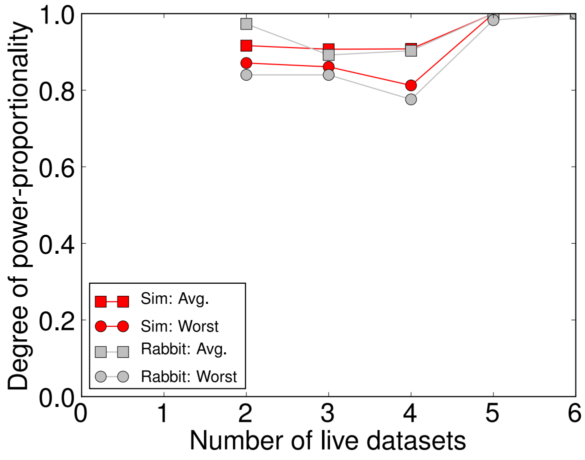
Table 2: Fair I/O scheduling provides isolation for app_A from app_B. As the number of instances of app_B increases, app_A running on default HDFS does worse, but app_A running with Rabbit based IO isolation is hardly affected. Each run was performed 3 times and the median value is reported.

4.3.2 *Fair I/O Scheduling*

The first policy that we discuss is a fair I/O scheduler. The scheduler guarantees equal sharing of I/O bandwidth among the datasets that are live. The I/O scheduler is not concerned with the applications that use the datasets. To demonstrate the effectiveness of the fair scheduler, two datasets A and B, with sizes 100GB and 40GB respectively, are stored in Rabbit with a replication factor of 3. The fair I/O scheduler is used to enforce fair sharing on the I/O bandwidth for the two datasets. Dataset A is used by a map-reduce application(app_A) using the Hadoop library, and dataset B is used by a distributed grep application (app_B) that directly reads data from the DFS. To test the fair I/O sharing, we increase the number of instances of the grep application. As can be seen from Table 2, the performance of the Hadoop application remains unaffected in the case of Rabbit but drops steadily as the number of instances of the grep application increases in the case of default HDFS. Thus, fair sharing can be used in the case of



(a) Total nodes:14; Primary nodes per dataset, p:2



(b) Total nodes:14; Total datasets:7

Figure 8: Multiple data set power proportionality, varying the total number of data sets and the number of active data sets. Both simulation and real-world experiments show very little interference, and the simulator is able to track the experimental results very closely.

a cluster where multiple users run their jobs on their own data, to provide a degree of performance isolation to each user.

ID	Entry time	Prio.	HDFS time t_0(s)	Rabbit time t_r(s)	Speedup (t_0/t_r)
1	1	1	393	558	0.704
2	20	3	182	98.4	1.85
3	200	5	152	94	1.61

Table 3: Priority scheduling allocates different numbers of servers to different jobs so that I/O bandwidth is allocated proportionally to priority. As before, reported values represent median of 3 runs.

4.3.3 *Priority Scheduling*

The second policy implements a priority-based scheduling policy, wherein jobs can be allocated different amounts of I/O resources. Consider the scenario shown in Table 3. We have 3 jobs with different arrival times and priorities. Job 1 is a Hadoop job that operates on a 100GB dataset, whereas Jobs 2 and 3 operate on 40GB and 20GB datasets, respectively. In the default HDFS case, there is no explicit allocation of I/O resources to each job/dataset. In the case of Rabbit, however, an allocation scheme divides the I/O bandwidth between the live datasets in a manner weighted by the priorities. For example, in the case shown, at $t = 25$, jobs 1 and 2 are running. The I/O bandwidth is therefore divided between those jobs in the ratio 1:3. This allows the priority-based scheduler to provide faster completion times for higher priority jobs. As shown in Table 3, jobs 2 and 3 obtain significant speedups at the expense of the longer-running, lower priority job 1. This scheduler would be useful in cases discussed by Isard et al. [11] where they note that almost 5% of the jobs in their data center ran for over 5 hours, but more than 50% of the jobs ran for less than 30 minutes. It would now be possible to decrease the completion time of shorter jobs at the expense of long-running jobs, if so defined by assigning priorities appropriately.

4.4 Fault Tolerance

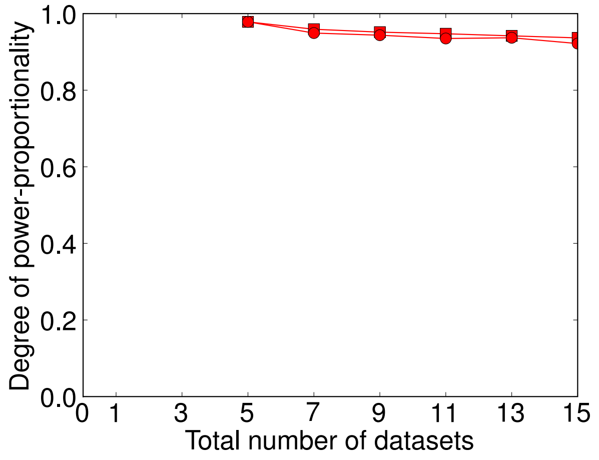
We evaluate the effects of the geared fault-tolerant data layout with an experiment that mirrors the simulation results from Section 3.5. This experiment sorts a 100 GB file on a 27 node cluster. This cluster is configured to have 6 primary, 15 secondary, and 6 tertiary servers. The secondary servers are configured in 5 gear groups, each with 3 servers. This means that, in the event of a primary failure, 5 secondaries will need to be activated.

The results of this experiment are shown in Figure 10. Only the time for the map phase of the job was measured, as this is the phase where all of the data is read from the storage servers. The remainder of the job is spent sorting, and writing the sorted results back.

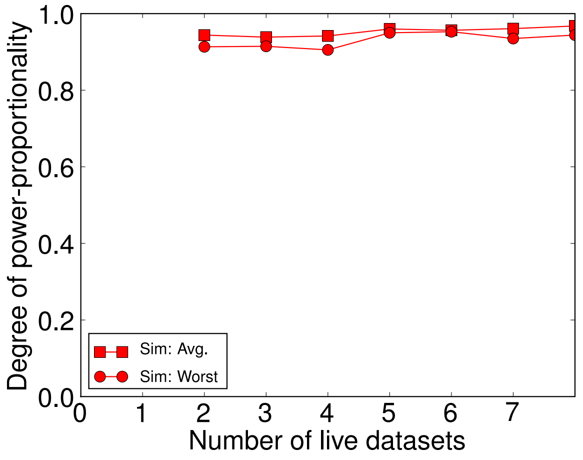
5. RELATED WORK

Power-proportionality was expressed as a desirable property for systems by Barroso and Hölzle [4]. There exists a large body of work on CPU power management techniques such as dynamic voltage and frequency scaling (DVFS). Owing to the increasing proportion of non-CPU power, there has also been significant research in the area of power management by turning off servers [7, 17] and consolidating load. Meisner et al. [13] discuss techniques to reduce server idle power through a variety of techniques. Guerra et al. [10] show the potential effectiveness of power proportionality in the data center. They describe the potential power savings of an ideally power proportional storage system using workload data collected from production systems. They also outline a number of techniques that may be used to achieve power proportionality.

A number of recent projects have attempted to bring power proportionality to the data center in general, and large scale file systems specifically. Leverich et al. [12] introduce the idea of a "covering set" of servers, analogous to



(a) Total nodes: 500; Primary nodes per dataset, p:20



(b) Total nodes: 500; Total datasets: 10

Figure 9: Simulation results showing interference between data sets for larger numbers of data sets. The interference is very low, though it does get worse as more data sets are added.

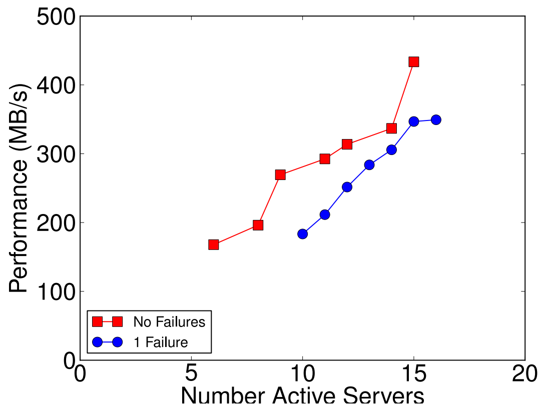


Figure 10: Performance of fault-tolerant layout with 6 primary servers and a gear size of 3. The number of active servers when a primary fails is 10: 5 primaries remain active, and 5 servers from the failed primary's recovery group are activated.

what we have been calling the "primary set". These servers satisfy the property that, if they are functioning, all data is available. We build on this work by exploring the effects of power scaling on efficiency.

Vasic et al. [21] also describe the problem of power scaling in HDFS and talk about an architecture to allow cluster services to collaboratively make power decisions. They use a data layout policy introduced for FAB [18] to maintain data availability when turning off up to two-thirds of the machines.

Weddle et al. [22] introduce the design of a power-aware RAID, using a geared scheme for managing different power settings. The block layout used by PARAID allows the array to be run in a variety of power modes and to efficiently migrate data as disks are activated. We adapt the technique of gearing to distributed file systems for providing power proportional fault tolerance.

Narayanan et al. [14] introduce the concept of write offloading for power management. When an application issues a write to a currently idle volume, the write may be redirected to an active volume. This technique increases the idle periods, allowing the system to save more power by spinning down idle disks. Everest [15] uses write offloading to avoid unusually high peaks in activity, allowing a system to be provisioned for less than the maximum request rate. Data is migrated to its target volume during periods of low activity to restore the desired data layout. In Section 3.3, we describe how these techniques can be used to offload write traffic from the primary servers onto others.

Thereska et al. [20] explore a power aware storage system using trace data collected from large-scale production services. In addition, Sierra introduces a technique for writing data while the system is in a low-power mode and migrating to the target layout in the background when the power mode is increased. Rabbit's data layouts complement Sierra's techniques for handling writes, providing for lower minimum-power settings, more fault-tolerant power-proportionality, and a mechanism for controlled sharing.

Rather than building on clusters of commodity servers, some researchers are exploring system designs specifically designed for particular classes of data-intensive computing. The FAWN [3] and Gordon [5] architectures use many nodes with low-power processors and flash storage to provide data-intensive computing and key-value indexing much more energy-efficiently, measured for example as queries per joule [3]. This is an interesting approach, which may also benefit from Rabbit's data-layout policies, but there is also value in exploring energy efficiency in commodity cluster-based storage.

6. CONCLUSIONS

Rabbit is a power-proportional cluster-based storage system. It uses novel *equal-work* data layout policies to support ideal power-proportionality for a wide range of power-performance states, power-proportional failure and recovery modes, and a building block for controlled sharing of available I/O capabilities. Experiments with a Rabbit prototype demonstrate its power-proportionality properties, and we show that the additional fault-recovery and sharing features come with minimal cost in power-proportionality. Rabbit's data-layout policies represent an important building block for shared, energy-efficient storage for cloud computing environments.

7. REFERENCES

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

[2] Pig. http://hadoop.apache.org/pig.

[3] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP '09: Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2009. ACM.

[4] Luiz A. Barroso and Urs Hölzle. The Case for Energy-Proportional Computing. *Computer*, 40(12):33–37, 2007.

[5] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications. In *ASPLOS '09: Proceeding of the 14th international conference on Arc hitectural support for programming languages and operating systems*, pages 217–228, New York, NY, USA, 2009. ACM.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. BigTable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):1–26, June 2008.

[7] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing Energy and Server Resources in Hosting Centers. *SIGOPS Oper. Syst. Rev.*, 35(5):103–116, December 2001.

[8] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI '04: Proceedings of USENIX Conference on Operating Systems Design and Implementation*, pages 137–150, 2004.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun T. Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[10] Jorge Guerra, Wendy Belluomini, Joseph Gilder, Karan Gupta, and Himabindu Pucha. Energy Proportionality for Storage: Impact and Feasibility. In *HotStorage '09: SOSP Workshop on Hot Topics in Storage and File Systems*, 2009.

[11] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276, New York, NY, USA, 2009. ACM.

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

[13] David Meisner, Brian T. Gold, and Thomas F. Wenisch. PowerNap: Eliminating Server Idle Power. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 205–216, New York, NY, USA, 2009. ACM.

[14] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

[15] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony Rowstron. Everest: Scaling down peak loads through I/O off-loading. In *OSDI '08: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, 2008.

[16] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposiu m on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[17] Cosmin Rusu, Alexandre Ferreira, Claudio Scordino, and Aaron Watson. Energy-Efficient Real-Time Heterogeneous Server Clusters. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 418–428, Washington, DC, USA, 2006. IEEE Computer Society.

[18] Yasushi Saito, Svend Frølund, Alistair Veitch, Arif Merchant, and Susan Spence. FAB: Building Distributed Enterprise Disk Arrays from Commodity Components. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 48–58, New York, NY, USA, 2004. ACM.

[19] Sorting 1PB with MapReduce, http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.

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

[21] Nedeljko Vasić, Martin Barisits, Vincent Salzgeber, and Dejan Kostic. Making Cluster Applications Energy-Aware. In *ACDC '09: Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, pages 37–42, New York, NY, USA, 2009. ACM.

[22] Charles Weddle, Mathew Oldham, Jin Qian, An-I Andy Wang, Peter L. Reiher, and Geoffrey H. Kuenning. PARAID: A Gear-Shifting Power-Aware RAID. *TOS*, 3(3), 2007.

[23] E.R. Zayas. AFS-3 Programmer's Reference: Architectural Overview. Technical report, Transarc Corporation, 1991.