Janus: Optimal Flash Provisioning for Cloud Storage Workloads

Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji
François Labelle, Nate Coehlo, Xudong Shi, C. Eric Schrock
{calbrecht,aamerchant,mstokely}@google.com,mhwaliji@gmail.com
{flab,natec,xdshi,eschrock}@google.com
Google, Inc.

Abstract

Janus is a system for partitioning the flash storage tier between workloads in a cloud-scale distributed file system with two tiers, flash storage and disk. The file system stores newly created files in the flash tier and moves them to the disk tier using either a First-In-First-Out (FIFO) policy or a Least-Recently-Used (LRU) policy, subject to per-workload allocations. Janus constructs compact metrics of the *cacheability* of the different workloads, using sampled distributed traces because of the large scale of the system. From these metrics, we formulate and solve an optimization problem to determine the flash allocation to workloads that maximizes the total reads sent to the flash tier, subject to operator-set priorities and bounds on flash write rates. Using measurements from production workloads in multiple data centers using these recommendations, as well as traces of other production workloads, we show that the resulting allocation improves the flash hit rate by 47–76% compared to a unified tier shared by all workloads. Based on these results and an analysis of several thousand production workloads, we conclude that flash storage is a cost-effective complement to disks in data centers.

1 Introduction

Disks are slow, and not getting much faster, even as their capacities grow: the random I/O operations possible per gigabyte stored on disk continues to decline. We can compensate for this by adding flash storage, which supports a much higher I/O rate per byte of storage capacity. Since flash is expensive per byte compared to disk, it is best to provision a relatively small amount of flash to store the most frequently accessed data.

Storage needs in a large cloud environment are often highly varied between different users and workloads [15, 23]. Hence, distributing the available flash capacity uniformly between the workloads is not ideal from either a performance or a cost perspective. Instead, we seek to leverage the differences between the competing users and workloads to optimize the provisioning of flash.

Our system, Janus, provides flash provisioning and allocation recommendations for both individual users and system administrators of large cloud data centers, where many users share the resources. Janus uses sparse traces, such as Dapper traces [22], to build a compact characterization of how effective flash storage is for different workloads. Where flash provisioning decisions are made by individual users, this characterization can be used to determine how much flash storage is cost-effective to purchase. For the case where resources are provisioned and allocated centrally by a system operator, we set up an optimization problem to partition the available flash between workloads so as to maximize the overall reads from flash and show how to solve it efficiently.

Janus recommendations are used by several production workloads in our distributed file system, Colossus [17]. We provide evaluations of the effectiveness of the recommendations from measurements on some of these workloads, and additional evaluations using traces of other production workloads. Our workload characterizations show that most I/O accesses are to recently created files. Based on this observation, files are placed in the flash tier upon creation and moved to the disk tier using FIFO or LRU eviction policies. Our results show that the recommendations allow 28% of read operations to be served from flash by placing 1% of our data on flash.

The three main contributions of this paper are:

- A characterization of storage usage patterns in a large private cloud focusing on the age of data stored and I/O rates to recently written data (§ 4).
- An optimization problem formulation for flash allocation to groups of files to maximize read rates offloaded to flash weighted by priorities and bounded by maximum flash write rates (§ 6).
- Experimental results from an implementation for the Colossus file system (§ 8).

2 Related Work

Several types of multi-tier storage systems [16] have been developed for memory, solid state drives, disk, and tape. These include Hierarchical Storage Management (HSM) [7, 12], multi-tier stores [24], multi-tier file systems [2], hybrid disk/flash storage [19], and extent-based enterprise volume management [24, 13]. Most include automated methods for migrating data between tiers based on I/O activity levels, performance requirements set by administrators, or explicit rules defined by users or administrators. However, none of these have focused on a distributed, cloud-scale deployment, which adds issues of provisioning policies and workload monitoring compatible with distributed management.

Several storage design tools, such as Minerva [3] and DAD [4], advocate principled, automated approaches to choose appropriate storage parameters for disk arrays based on workloads and desired availability characteristics. However, these tools typically provide only coarse-grained recommendations about RAID levels for storage volumes, unlike the data placement decisions for different files in a multi-tiered cloud storage environment described in this paper.

Studies on the distributed file system in Sprite [5] and the local file system in 4.2 BSD [20] showed the utility of characterizing user activity, access patterns, and file lifetimes when evaluating caching strategies. Blaze [6] analyzed access patterns affecting caching in a distributed file system using traces of I/O activity obtained by monitoring storage related remote procedure calls (RPCs). We similarly monitor storage RPCs in our distributed file system, but we also needed to use sampling and other statistical techniques due to the system scale.

TIP [21] used explicit hints of future I/O accesses provided by the application programmer to determine which data to prefetch and when. Janus does not rely on the explicit programmer action of adding hints to the API usage of the system. Instead, we predict the cacheability of different user workloads automatically from online measurements of past usage. Kroeger [14] predicts file access patterns in the context of prefetching at the Linux kernel level by using the sequence of past accesses; however, it is not clear how it could be extended to the distributed case.

Our approach is most closely related to the work of Narayanan et al. [18], which analyzed several enterprise workload traces to evaluate the economic feasibility of replacing disks with flash storage. We focus on a larger cloud storage environment, develop an algorithm for making good allocation choices between different workloads, and reach significantly different conclusions about the effectiveness and economics of using flash in this manner.

3 System Description

Janus provides flash storage allocation recommendations for workloads in a distributed file system, such as Colossus, in a large private cloud data center. The underlying storage is a mix of disk and flash storage on distinct *chunkservers*, structured as separate tiers. Upon creation, files may be placed in the flash tier, and later moved to disk using a FIFO or LRU policy. We use this *insertion on write* mechanism rather than the *insertion on read* used in most caches because it is more suitable for our system. The distributed nature of file systems like GFS and Colossus makes insertion on read policies more expensive than insertion on write for some metrics we intend to optimize, in particular the volume of read activity. Because data access occurs directly between chunkserver nodes and clients, and not every chunkserver node contains flash capacity, an insertion on read policy that does not rely on the client for write back must perform an additional read in order to populate the data into flash storage. Additionally, the write back into flash storage can not be assumed to be instantaneous as the operation requires reading data from disk, transferring across a local network link, and finally a write into the flash media.

Many users and applications may use this system, either directly, or through higher level storage components such as Bigtable [8]. The flash tier can be partitioned between workloads. The main scenario we consider is where the system operator has a fixed amount of total flash available in the system, and wants to maximize the fraction of reads offloaded to flash storage; however, in some cases, it may be preferable to offload high-priority workloads.

Workloads can correspond to users, applications, or specified groups of files. For example, an application may have separate logging, data, and metadata components, and these could be different groups. In structured data, a table or a set of columns may be a group. Exactly how the workloads are formed is outside the scope of this paper; we just assume that these groupings exist, perhaps manually created. We may associate a *priority weight* with each workload; the higher the priority weight, the more important it is to accommodate its reads from flash storage. The precise mechanism for choosing workload weights is again outside the scope of this paper, but for example, the administrator could assign different weights to different workload types. Our goal is to determine automatically how to divide the available flash between the workloads to optimize the reads from flash.

The allocation recommendations are made by an offline optimization solver that runs periodically to adjust to changes in the workload behaviors and the available flash storage. A key input to the solver is a compact representation of both the age of the data stored in each

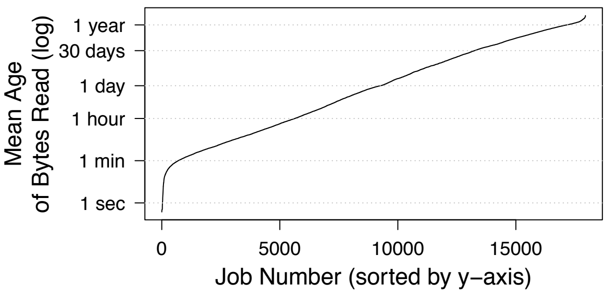

Figure 1: Mean age of bytes read differs significantly by user.

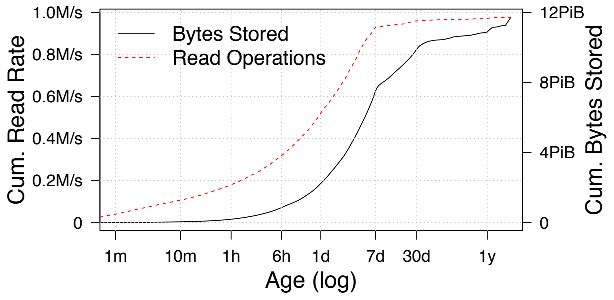

Figure 2: Cumulative distribution function of the bytes stored, and read operations sorted by the (FIFO) age of the data for a particular workload. These CDFs are a graphical representation of the histograms collected as inputs to the cacheability curves, which are different for each user and used in the optimization formulation. 50% of the data stored by this particular user is less than 1 week old, but that corresponds to over 90% of the read activity.

workload group, and the read rate of the data by age. These are obtained by scanning the file system metadata and sampled traces of I/O activity.

The operation of Janus can be broken into three steps:

- Collection of input data about the age of bytes stored and age of data accessed for different workloads to generate a characterization of how cacheable each workload is (§ 4).
- Solving an optimization problem to allocate flash amongst the workloads (§ 6).
- Coordination with the distributed file system to place data from different workloads on flash using the computed write probabilities and flash sizes from the solver (§ 8).

4 Workload Characterization

Storage in our data centers is shared between thousands of users and applications. Applications include content indexing, advertisement serving, Gmail, video processing, as well as smaller applications, such as MapReduce jobs owned by individual users. A large application may have many component jobs. The workload characteristics and demands of jobs in data centers are typically highly varied between users and jobs. Figure 1 shows the variation of mean read age over different jobs in our data centers. All read ages are well represented: there are jobs accessing very young (1 minute old) to very old (1 year old) data. However, different jobs also have very different read hotness, as shown in Figure 4, so we cannot conclude that the aggregate reads are evenly distributed over data of different ages. Instead, we need to define a metric that lets us compare how many read operations would be served by flash storage for a given flash allocation to that workload.

4.1 Cacheability Functions

The cacheability function (which we define more formally below) tells us the rate of read hits we are likely to get for a workload if we allocate it a given amount of flash. To compute this for FIFO eviction, we need two inputs for each workload: how much data there is of a given age, and how many reads there are to files of a given age. For LRU eviction, the corresponding two inputs are the amount of data with a given temporal locality and the rate of reads to files with that temporal locality.

We define two age metrics: FIFO age and LRU age, which are used with the corresponding eviction policies (although we will just say "age" where the disambiguation is not needed). The FIFO age of a file (and of all the data in the file) is the time since the file was created. The LRU age of a file, which is a measure of the temporal locality of its reads, is approximately the maximum time gap between reads to the file since it was created (see Section 8.7 for a precise definition).

Obtaining the distribution of FIFO age is straightforward: we scan the file system metadata, which includes the create time of each file, to build a histogram of the FIFO age of bytes stored for each group. To build a histogram of the read rates of data by FIFO age, we need to look at the read accesses, which we obtain from traces. Since the read rate in the data centers is enormous, it is not practical to consider every read to the data in each workload. Instead, we sample the reads from every job using Dapper [22, 9], an always-on system for distributed tracing and performance analysis. Dapper samples a fraction of all RPC traffic and, by looking at the age of the requested data at the time of each RPC, we can populate a second histogram of the number of read operations binned by age of the data read. Crucially, each of these two histograms has the same bucket boundaries for data age, which later lets us join the histograms.

Computing the corresponding histograms for LRU age is similar, except that computing the LRU age requires the time-gaps between read operations to a file. Dapper traces do not suffice in this case, since not every I/O to

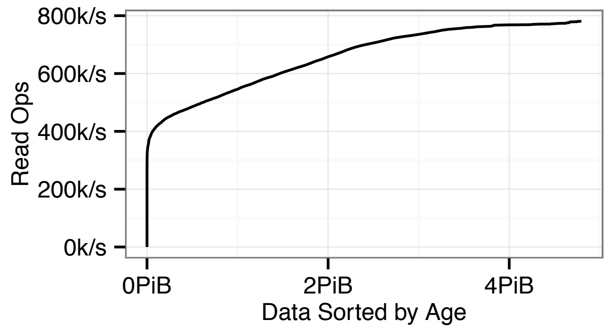

Figure 3: The number of read operations for a given amount of the youngest data (by FIFO age) for a particular user.

a file is captured. We built instead a distributed tracing mechanism that samples based on the file identifier and captures every I/O for the files so selected.

The two input histograms of data age and read age for a specific workload can then be combined to construct a cacheability function.

Definition (Cacheability Function): For a workload the cacheability function ϕ maps the amount of flash allocated to the workload to the rate of reads absorbed by the flash storage. In particular, $\phi(x)$ gives the number of read operations that go to the youngest x bytes of data.

Figure 3 shows an example of a cacheability function computed by joining the histograms of read rate and data size by age in Figure 2. Joining the histograms is simple because the age bins are the same. From the histograms for a specified workload we derive the cumulative function f giving the amount of data younger than a certain age, and the cumulative function g giving the read operations to files that are younger. Essentially, for each flash allocation x, we can look up the age $f^{-1}(x)$ of the files that can be stored (assuming the youngest are stored) and then look up the rate of read hits for files of that age or younger:

$$\phi(x) = (g \circ f^{-1})(x) = g(f^{-1}(x))$$

We compute the cacheability function by linear interpolation between the bins, and hence the function is piecewise linear, a fact we later use in the optimization. Assuming that these distributions are stationary, the composition gives us the read hit rate for the flash allocation. Also, because of the way we separately defined file age for FIFO and LRU eviction, this method works in both cases.

5 Economics and Provisioning

Narayanan et al. [18] argued that replacing disk with flash storage was not cost-effective. Prices for flash have fallen considerably since then, but has this conclusion

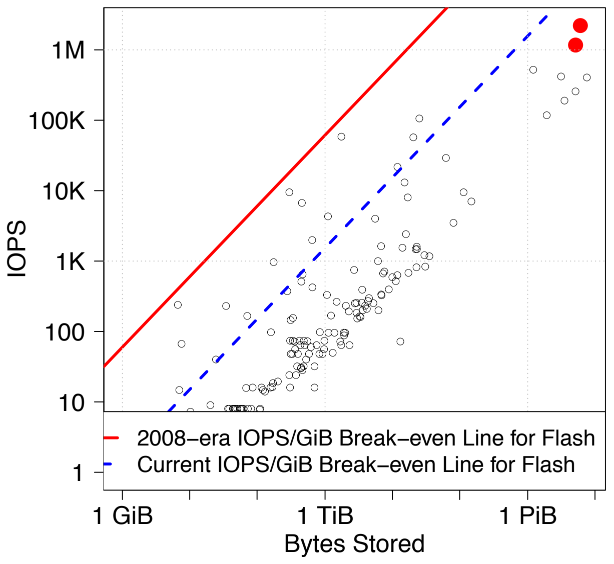

Figure 4: Peak IOPS and capacity requirements for user workloads in a shared data center. IOPS is the 95th percentile over 10 minute intervals. Workloads above the break-even line would be cost-effective to store entirely on flash. The two filled red dots are for the workloads in Table 1, and come from other data centers. The lower dot is for the workload in Figure 3.

changed? To analyze this, following Narayanan, we find the *break-even point*, which is the IOPS/GiB threshold determining whether a workload would be cheaper on flash storage or on disk. This threshold can be derived from the IOPS/\$ of disk, I_d, and the GiB/\$ of flash, G_f, since a workload with I IOPS and G GiB will cost G/G_f on flash and at least I/I_d on disk (more for cold data). Therefore, workloads with IOPS/GiB greater than I_d/G_f are better served from flash, and by using a disk with high I_d for this cutoff, we are being conservative in recommending workloads to go entirely on flash.

For our example IOPS/\$ efficient retail drive, we use the Seagate Savvio 10K.3, which costs around \$100. The disk specifications [1] indicate an IOPS capacity around 150 ((seek time+avg latency)$^{-1}$), or 1.5 IOPS per dollar for disk. On the other hand, recent news reports [10] indicate that we can get about 1 GiB of flash per dollar; together these give a break even point of 1.5 IOPS per GiB, which is much smaller than the 2008 value $\approx$ 60. As displayed in Figure 4, we find that, at least for some workloads, it is cost-effective to place all data in flash. Even for other workloads close to the break-even point, using flash may be justified by the resulting improvement in latency.

In addition, many workloads could benefit from putting their youngest data on flash using Janus. We now consider how much flash is cost-effective for an individual workload. For a workload with read operations rate rate$_r$, write operations rate rate$_w$, capacity size c, and

Workload	1	2
Data size (PiB)	5.2	6.1
Access rate (k ops/sec)	1172	2214
Janus Savings (%)	29	12
Janus Flash (%)	0.42	2.1

Table 1: Storage demands and savings from a price optimization using Janus, which correspond to solid red dots in Figure 4. The savings is over the best all-disk or all-flash solution. The flash $\%$ is the percentage of the user data in flash.

cacheability function $\phi()$, a disk $(\text{IOPS},\ \text{GiB}_{disk})$ demand of $(\text{rate}_r + \text{rate}_w,\ d)$, could be replaced with a disk + flash $(\text{IOPS}_{disk},\ \text{GiB}_{disk},\ \text{GiB}_{flash})$ demand of

$$(\text{rate}_r + \text{rate}_w - \phi(x),\ d - x,\ x)$$

To determine the amount of flash, x, that a user should purchase, and their benefit from using the system, we impose a pricing structure, then have each user purchase flash to minimize costs. We avoid pricing complications arising from the balance of cold and hot data in a shared storage system, and put ourselves in the IOPS constrained framework where we sell disk based entirely on IOPS, so that cost is determined by

$$\text{cost}(x) = \frac{(\text{rate}_r + \text{rate}_w - \phi(x))}{I_d} + \frac{x}{G_f} \qquad (1)$$

and we note that the optimization of cost is simplified by the fact that ϕ is piecewise linear between histogram buckets.

In Table 1 we consider this optimization for some workloads and display the price savings, along with the percentage of data that goes on flash, in the optimal configuration. We note that while workload 2 is hotter on average, workload 1 gets a greater benefit from a smaller amount of flash because of its steep cacheability curve (Figure 3).

6 Optimizing the Flash Allocation for Workloads

We now describe how we determine the best flash allocation for each workload, given the cacheability functions derived in Section 4. Specifically, we seek to maximize the aggregate rate of read operations served from flash subject to a bound on the total available flash. The workloads may have different priority weights, in which case we maximize the aggregate weighted rate of reads from flash.

We assume that the cacheability functions are piecewise linear and concave. As mentioned previously, the piecewise linear assumption always holds since we compute the function by linearly interpolating between a finite number of points (corresponding to the bins of the histogram from which we derive it). The concavity assumption is equivalent to assuming that the read rates for each workload's data decrease monotonically with increasing data age. This assumption holds usually, but not always. We will show in the next section how to relax the assumption where it matters.

Weighted Max Reads Flash Allocation Problem Instance:

- A set of workloads; for each workload i is given the total data d_i, the cacheability function as a piecewise linear function $\phi_i : [0, d_i] \to \mathbb{R}$, and a priority weight ρ_i.
- A bound on the total flash capacity F.

Task:

Find for each workload i the allocated flash capacity x_i, $0 \le x_i \le d_i$, maximizing the total priority weighted flash read rate $\sum_i \rho_i\, \phi_i(x_i)$, and subject to the constraint of the total flash capacity $\sum_i x_i \le F$.

Let the segments of the piecewise linear function $\rho_i\, \phi_i$ be $a_{i,j} + b_{i,j}\, x$ for $j = 1, \ldots, n_i$. Since ϕ_i is concave, $\rho_i\, \phi_i$ can be expressed as the minimum of the functions corresponding to its linear segments:

$$\rho_i\, \phi_i(x) = \min_{1 \le j \le n_i} \{a_{i,j} + b_{i,j}\, x\}$$

By replacing $\rho_i\, \phi_i(x)$ with the variable y_i, we transform the task into a linear program:

$$\begin{array}{lll} \max & \sum_i y_i & \\ \text{s.t.} & y_i \le a_{i,j} + b_{i,j}\, x_i & \text{for each workload } i \\ & & \text{and each segment } j \\ & \sum_i x_i \ \le\ F & \\ & 0 \ \le\ x_i \ \le\ d_i & \text{for each workload } i \end{array} \qquad (2)$$

This optimization problem can be solved with an LP solver. We solve it directly as explained at the end of the next section.

7 Optimization with Bounded Write Rates

Limiting flash write rate is important to avoid flash wear out and also reduces the impact of flash writes (which are slow) on read latencies. We now describe how to allocate flash so as to maximize the reads from flash while limiting the write rate to flash. We also show how to approximately relax the concavity assumption on the cacheability function. The cacheability function for a workload may be non-concave if the read rate increases some time after it is created, for example, if there is a workload that begins processing logs with some delay after they are created.

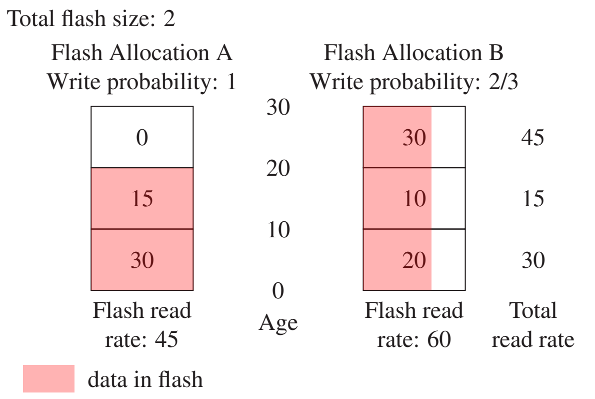

Figure 5: Example for non-concave cacheability and fractional write probability: Data blocks and read rates of a workload for different age ranges are shown at steady state. The workload has one block of data with age between 0 and 10 and a read rate of 30, a second block of data with age between 10 and 20 and a read rate of 15, and a third block of data with age between 20 and 30 and a read rate of 45. Storing all data younger than age 20 in flash (highlighted) gives a hit rate of 45 (left). With a write probability of $2/3$, we place less new data in the same flash but keep it longer, until age 30 (right). This captures the higher read rate for data between ages 20 and 30, for a total flash hit rate of 90 * 2/3 = 60. The write rate to flash also decreases by $1/3$.

We only consider insertion into flash upon write, so that the write rates per workload are independent of the flash allocation. For simplicity, we also ignore priority weights here, but this extension is straightforward.

The flash write rate can be controlled either by limiting the workloads that have data in flash or by writing only a fraction of each workload's new data into flash. We implement the latter by setting a write probability, and for each new file, deciding randomly with that probability whether to insert it into flash. Figure 5 shows an example with one workload where decreasing the write probability decreases the flash write rate *and* increases the flash read hits. This is only possible if the cacheability function is non-concave.

In general, if the workload i has a flash capacity x_i and a write probability p_i the data can stay in flash for as long as if the workload has a flash capacity of $\frac{x_i}{p_i}$ but all new data is written to flash. Hence, the flash read rate for the workload i with cacheability function ϕ_i is $p_i\,\phi_i(\frac{x_i}{p_i})$.

Bounded Writes Flash Allocation Problem
Instance:

- A set of workloads; for each workload i is given the total data d_i, a continuous piecewise linear cacheability function $\phi_i : [0, d_i] \to \mathbb{R}$, and a write rate w_i.
- A bound on the total flash write rate W.
- A bound on the total flash capacity F.

Task:

Find, for each workload i, the allocated flash capacity x_i, $0 \leq x_i \leq d_i$ and the flash write probability p_i, $0 \leq p_i \leq 1$, maximizing the total flash read rate $\sum_i p_i\,\phi_i(\frac{x_i}{p_i})$ and subject to the constraint of the total flash write rate and total flash capacity. Formally:

$$\begin{array}{lrcll} \max & \sum_i p_i\,\phi_i\left(\frac{x_i}{p_i}\right) & & & \\ \text{s.t.} & \sum_i p_i\,w_i & \leq & W & \\ & \sum_i x_i & \leq & F & \\ & 0 \leq x_i & \leq & d_i & \text{for each workload } i \\ & 0 \leq p_i & \leq & 1 & \text{for each workload } i \end{array} \tag{3}$$

While the problem has linear constraints, the objective is not linear. Our approach is to (a) relax the constraint on the write rate via Lagrangian relaxation; (b) remove the dependence on the write probability p_i in the objective function; (c) linearize the objective; and (d) solve the resulting linear program with a greedy algorithm.

We relax (remove) the write rate bound $\sum_i p_i w_i \leq W$ and change the objective function by subtracting the write rate with a write penalty factor $\lambda \geq 0$:

$$\sum_i p_i\,\phi_i\left(\frac{x_i}{p_i}\right) - \lambda\,p_i\,w_i \tag{4}$$

An optimal solution for the relaxed problem with a total write rate equal to the bound (i.e., $\sum_i p_i w_i = W$) is an optimal solution of the original problem (3). Proof: If there is a better solution for the original problem (3), its read rate is higher but its write rate cannot be larger. Hence, this solution is also a better solution for the relaxed problem, which contradicts the optimality.

Since the total write rate found by the relaxed optimization decreases monotonically with increasing λ, we can find the best λ, where the total write rate closely matches the bound, using binary search.

Since the constraints on the write probabilities p_i are independent of the other variables, we can remove the dependence on the write probabilities as follows. Let $h_i^\lambda(x)$ represent the contribution of workload i with allocated flash size x to the objective (4) when maximized. Then:

$$\begin{aligned} h_i^\lambda(x) &= \max_{0\leq p\leq 1} p\,\phi_i\left(\frac{x}{p}\right) - \lambda\,p\,w_i \\ &= \max_{z\geq x} \frac{x}{z}\,(\phi_i(z) - \lambda\,w_i) \\ &= x \max_{z\geq x} \frac{\phi_i(z) - \lambda\,w_i}{z} \end{aligned}$$

Since the function ϕ_i is continuous and piecewise linear, $(\phi_i(z) - \lambda\,w_i)/z$ is monotonic with z in each segment. Hence the above maximum can be found by evaluating it only at the breakpoints of ϕ_i. By processing the breakpoints of ϕ_i in decreasing x-coordinate the function $h_i^\lambda(x)$ can be computed in linear time.

Next, we linearize the resulting objective $\sum_i h_i^\lambda(x_i)$. h_i^λ is concave if ϕ_i is, which is usually the case because read rates decline with age. If not, we replace it with its concave upper bound by removing some breakpoints of the piecewise linear function. We argue later that this has only a small impact on the optimality of the result. As in the previous section, we rewrite $h_i^\lambda(x)$ as the minimum of the linear functions corresponding to its segments and get a linear program that has the same form as (2).

Finally, we solve this linear program with a greedy algorithm. We start with the solution $x_i = 0$, $y_i = 0$ for each workload i and then successively increase the allocated flash x_i of the workload that has the highest ratio of increase in the objective function to flash allocation, as long as flash can be allocated. Except for the last incremental allocation, the flash allocation to each workload corresponds to a breakpoint of its cacheability function. The algorithm has a runtime complexity of $O(n\,k\,\log k)$ where n is the maximum number of pieces of the piecewise linear functions ϕ_i and k is the number of workloads.

The result is optimal if h_i^λ is concave. If not, we can show that the error in the objective value due to the concave approximation is bounded by the objective increment of the last step. Hence, we are close to optimality if the last incremental flash allocation is small. This is certainly the case if the workload is partitioned into many small workloads, which is, in any case, preferable for optimal allocation.

8 Evaluation

In this section, we evaluate the effectiveness of the algorithms described in the previous sections on production storage workloads in Google data centers. Section 8.1 describes the production environment, and Section 8.2 introduces the datasets and terminology used for the evaluation. The remainder of the section evaluates the recommendations produced by Janus both on production workload deployments that used the recommendations and on traces of other production workloads.

8.1 File Placement in Colossus

Colossus (the successor of GFS [11]) is a distributed storage system with multiple master nodes and many chunkservers that store the file data. File system clients create new files by a request sent to a master node, which allocates space for it on chunkservers it selects. We evaluated Janus in a Colossus system extended as follows.

When a file is created, a Colossus master node decides, based on the amount of flash space available for the corresponding workload and the write probability assigned to it, whether it should be placed on disk or on flash, and accordingly allocates space. Eviction from flash is designed to take advantage of the already existing file maintenance scanner process. The file is tagged with an eviction time (TTL), which is computed from the flash allocated to that workload and the workload's write rate. The scanner process periodically checks whether the file has exceeded its eviction time, and if so, moves it to disk. The eviction time (TTL) in its current implementation is not updated after it is set, effectively producing an *approximate FIFO* eviction policy. An arriving file creation request sometimes finds the flash storage full; in this case, the master will write it to disk, regardless of whether it would otherwise have chosen to write it to flash.

8.2 Datasets and definitions

In the remainder of Section 8, we evaluate Janus based on several datasets.

A Colossus *cell* is a separate instance of the Colossus system. Separate cells are typically located in different facilities. Each cell has its own masters, chunkservers, and files, and each cell independently manages user quota.

The first three datasets come from multi-user cells, with workloads corresponding to different users of the cell.

Dapper A 37-day Dapper sample of read-write activity over 10 cells. The first 30 days are used for training (computing the cacheability functions), and the last 7 days are used for evaluation.

Janus Deployment Data from limited deployments of production workloads using Janus recommendations in 4 cells. In these deployments, flash was assigned only to a single workload. The training period consisted of a 30 day of Dapper sample prior to the deployment.

Multi-User Cell A 1-week trace of all read-write activity to a 1% sample of files in a single cell. The 6th day is used for training, and the 7th day is used for evaluation. The first 5 days are used in Section 8.7 to determine whether a file is cached by LRU. This cell had 407 workloads.

The last dataset comes from a cell where all activity comes through Bigtable. Files are separated into workloads based on tokens that Bigtable encodes in the file name for different tables and columns.

Single-User Cell A full trace of read-write activity in a single-user Colossus cell for 30 minutes. The first 15 minutes are used for training, and the second 15 minutes are used for evaluation. The cell had 541 workloads, and contained over 10,000 machines. A configuration change was needed to collect the data, leading to the short duration of the trace, but it contains adequate data because of the size of the cell and the fact that the trace is not sampled.

The *read rate* for each workload is the number of read operations per second, excluding in-memory cache hits. The *flash read rate* is the number of read operations per second that is served from flash. The *flash hit rate* is the flash read rate as a percent of the read rate. In some cases, we report the *normalized flash hit rate*, which is the flash read rate for a workload as a percent of the total read rate among all workloads in the cell. In particular, the cell-wide flash hit rate is the sum over all workloads of the normalized flash hit rate.

The *size* of a workload is the logical number of bytes stored, excluding overhead from replication or erasure coding. Analogous to the terminology for read rates, we also have *flash size*, *flash size percentage* and *normalized flash size percentage*.

The *write rate* of a workload is the number of bytes per second of new data written. Again, this excludes overhead from replication or erasure coding. Again, we also have *flash write rate* and *flash write percentage*.

Given a cell-wide flash size, or equivalently a flash size percentage, we form an *allocation* of flash to different workloads using the optimization. The allocation consists of a Flash Size and a Write Probability for each workload in the cell. This allocation is used in the evaluation period to compute various metrics of interest, such as flash hit rates.

8.3 Does the Past Predict the Future?

Our optimization is based on sampled historical data. Here, we investigate the stability of estimated per-workload flash hit rates between training and evaluation periods in the Dapper dataset. In Section 8.4, we will consider how well estimated flash hit rates correspond with values from an actual deployment.

We chose 10 cell-wide flash size percentages between 0.1% and 10%. For each flash size percentage and each cell, we optimized the allocation of flash to workloads. Figure 6 uses these allocations to plot per-workload flash hit rates in the evaluation period against those in the training period. The figure shows that flash hit rate during evaluation is typically within 7% of the flash hit rate during training. This range of variability is small enough for the resulting system to be effective.

8.4 Janus Deployment

Due to the Colossus's use of approximate FIFO (described in Section 8.1), we must compute eviction TTLs

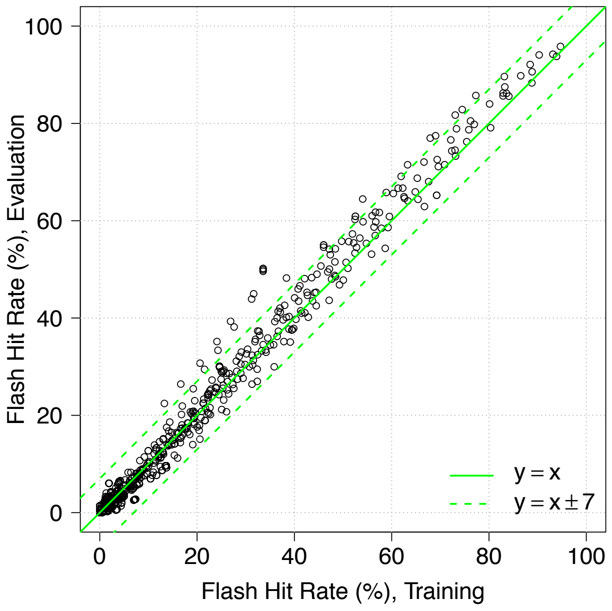

Figure 6: Flash hit rate during training and evaluation periods, estimated from the Dapper dataset. Each point represents a single workload in a single cell with a given cell-wide flash size percentage.

from each workload's allocation. Janus computes the TTLs using file age histograms from the training period. However, the file age distribution may change between training and deployment. For example, a workload may start writing new data at a high rate, or it may exhibit peak-to-trough variability not captured in histograms averaged over the 30-day training period. In these cases, using fixed TTLs may cause the workload to exceed its allocated flash size, and Colossus will write new files to disk until flash usage decreases. Hence, a workload's actual flash usage can fluctuate over time.

Figure 7 shows flash usage for a single workload over two days in one cell. The workload's allocated flash size was 100 TiB. Each day, actual flash usage fluctuates from 45 TiB to 100 TiB due to peak-trough variations. We accommodated this variation by decreasing the allocated flash size so as not to exceed the actual allocation.

Figure 7 also shows the workload's flash read rate during the period. On average, we get around 30k flash read ops/sec, with a peak of more than 40k flash read ops/sec. From the 30 day training period, we predicted a flash read rate of 33k flash read ops/sec.

Table 2 shows results for this workload over deployments in four different cells. The average of estimated and measured flash hit rates over those cells were 22.8% and 23.5% respectively, a 3% difference. For cell A, the measured flash hit rate (27%) was significantly higher than the estimated value (17%), partly because we manu-

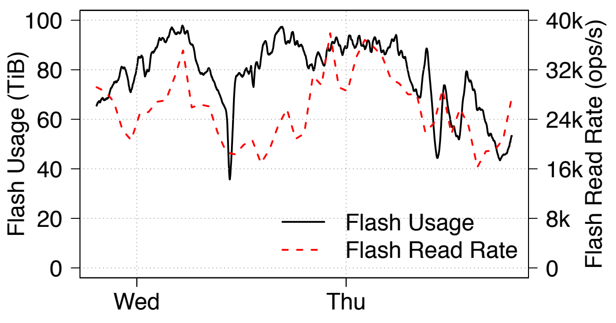

Figure 7: Flash usage and Flash read rate for one workload over two days after Janus deployment.

Cell	workload size	allocated flash size	average flash usage	estimated flash hit rate	measured flash hit rate
A	3.26 PiB	80 TiB	62 TiB	17%	27%
B	3.34 PiB	100 TiB	72 TiB	29%	31%
C	3.47 PiB	60 TiB	50 TiB	19%	16%
D	3.26 PiB	100 TiB	63 TiB	26%	20%
Avg	3.33 PiB	85 TiB	62 TiB	22.8%	23.5%

Table 2: Janus performance with one workload in four cells.

ally adjusted the parameters to maximize the space usage and allow the group to hit the quota.

8.5 Comparing Alternative Allocation Methods

In this section, we compare alternative methods for generating flash allocations. *Optimized FIFO* allocation uses the methods described above. We can also set per-workload flash size *proportional to read rate* or *proportional to size*. Both read rate and size are the average usages measured over the training period. Lastly, we can assign flash size such that the eviction TTL is the same for all workloads. This is effectively a *single FIFO* for all workloads. These and all subsequent comparisons are made using trace-based analysis rather than direct measurement, since Janus was only deployed with optimized FIFO eviction (denoted *Opt FIFO* in the tables).

Table 3 shows cell-wide flash hit rates for the single and multi-user cells. In the multi-user cell, the flash hit rate improves from 19% to 28% when changing from single FIFO to optimized FIFO, representing a 47% improvement. In the single-user cell, the relative improvement was even larger — from a 42% hit rate to 74%, a 76% relative improvement.

Especially in the single-user cell, optimized allocation outperforms the other methods. Table 4 shows that the poor performance of non-optimized methods in the single-user cell is due to allocating large amounts of flash to workload 117. This workload comprises 10% of the cell's read rate, but 43% of the cell's size. Optimized allocation assigns no flash to this workload, since other workloads provide a better read rate to size ratio.

Dataset	Multi-User	Single-User	Single-User
Flash Size (%)	1.0%	5.3%	5.3%
Additional Constraints			No flash for workload 117
Opt FIFO	28%	74%	74%
Prop. Read Rate	26%	64%	64%
Single FIFO	19%	42%	45%
Prop. Size	14%	15%	21%

Table 3: Flash hit rates achieved by 4 different allocation methods for the single and multi-user cells. The cell-wide flash size percentages were 5.3% for the single-user cell and 1.0% for the multi-user cell.

The last column of Table 3 shows that the flash hit rate under Single FIFO and Proportional to Size improves if we constrain workload 117 to receive no flash. However, Proportional to Read Rate does not improve, as removing workload 117 exposes the next few workloads that have a high read rate to older data.

The improvement between single FIFO and optimized FIFO in the multi-user cell can also be attributed to a single workload. This is discussed further in Section 8.7.

Normalized Flash Hit Rate (%)

Workload	Opt FIFO	Prop Reads	FIFO	Prop Size
1	11.8	10.8	8.1	3.5
2	7.9	7.9	4.2	1.3
3	7.7	5.5	6.0	2.2
4	7.4	7.4	3.8	1.1
5	5.7	5.7	2.2	0.5
9	4.0	4.0	4.0	0.3
10	3.9	4.2	4.2	0.3
117	**0.0**	**0.6**	**0.9**	**1.0**
Others	25.4	17.6	8.5	4.6
Total	73.7	63.8	41.9	14.7

Normalized Flash Size Percentage (%)

Workload	Opt FIFO	Prop Reads	FIFO	Prop Size
1	0.82	0.59	0.22	0.04
2	0.08	0.41	0.02	0.00
3	1.19	0.62	0.71	0.17
4	0.09	0.38	0.02	0.00
5	0.04	0.31	0.01	0.00
9	0.01	0.20	0.01	0.00
10	0.00	0.21	0.01	0.00
117	**0.00**	**0.25**	**0.99**	**2.26**
Others	3.02	2.27	3.26	2.77
Total	5.25	5.25	5.25	5.25

Table 4: Flash hit rates and size for selected workloads in the single-user cell. Workloads are numbered in decreasing order of flash read rate under optimized FIFO allocation.

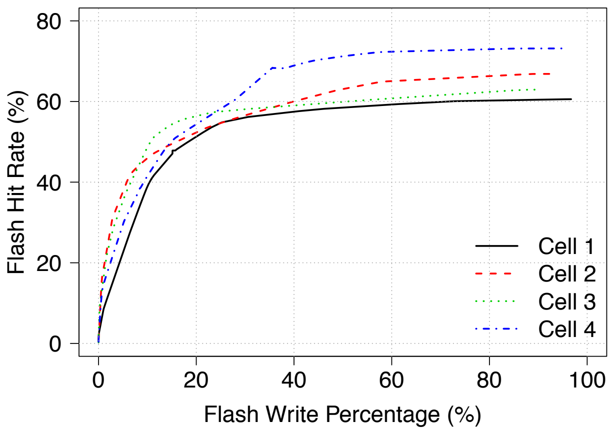

Figure 8: Flash hit rate for given bounds on the flash write percentage for four cells in the Dapper dataset.

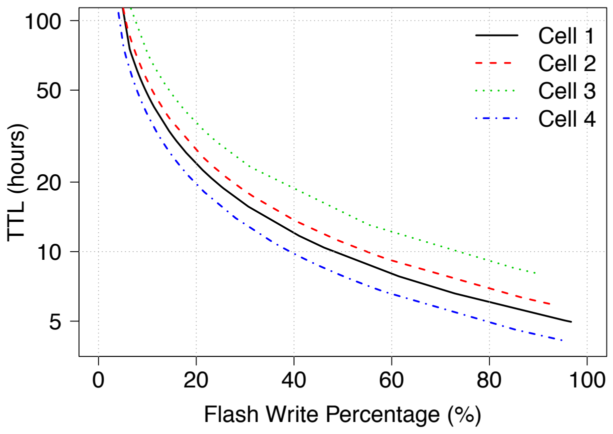

Figure 9: Average TTL of data written to flash for given bounds on the flash write percentage for four cells in the Dapper dataset.

8.6 Impact of Bounded Flash Write Percentage

In Section 7 we showed how the flash write percentage can be bounded at the cost of a lower flash hit rate.

Figure 8 shows the optimized flash hit rate for various bounds on the flash write percentage. The rightmost point on each curve corresponds to unbounded flash write percentage. In each cell, the optimized unbounded value is above 90%. As we decrease the bound, the flash hit rate decreases slowly at first, and it decreases quickly once the bound falls below 60%.

Figure 9 shows the average TTL of the new data written to flash for the same cells and the same flash allocation solutions. As the bound on the write rate is tightened, less data is written to flash but it stays there longer.

8.7 Evaluation of LRU Eviction

We have so far seen the performance of Janus with FIFO eviction. We now turn to an evaluation with LRU eviction.

LRU Cacheability Functions and Censoring

In Section 4, we briefly described cacheability functions for LRU eviction. We make this description more formal here.

A file will be in the cache if the maximum gap between reads is lower than the TTL. We re-define the notion of *age* to reflect this heuristic. The LRU age of a file at time t is

$$\text{Age}(t) = \max(t_1 - t_0, ..., t_n - t_{n-1}, t - t_n)$$

where t_0 is file creation time, and $t_1, ..., t_n$ are the times of the n reads in interval $[t_0, t)$. The smaller the LRU age of a file is, the more temporal locality its reads have. The cacheability function, $\phi(x)$, gives the flash read rate if the x bytes with the lowest LRU age are placed on flash.

To compute the age of a file at t, we require a full trace of read operations during $[t_0, t)$. In many cases, the full trace is not available. Suppose the trace is available only during $[t_S, t)$, with $t_S > t_0$. The resulting read age measurement is censored.

We deal with censoring by considering the two extremes. *Upper bound age* assumes that there were no reads between t_0 and t_S. *Lower bound age* assumes that there was continuous read activity between t_0 and t_S, so that $\text{Age}(t_S) = 0$. The upper bound of the cacheability function is obtained by using upper bound age for size and lower bound age for read rates, and vice versa for the lower bound of the cacheability function.

Evaluation of LRU using Multi-User Cell Dataset

Figure 10 shows the cacheability function for a single FIFO / LRU. With 1% flash size percentage, the flash hit rates are 19% for a single FIFO and 36%–40% for a single LRU. The marked points allocate flash to workloads using optimized FIFO / LRU. The flash hit rates are 28% for optimized FIFO and 44%–48% for optimized LRU.

Table 5 shows normalized flash hit rates for various workloads. Most of the improvement between FIFO and LRU can be attributed to the cacheability of workload 1, which is a Bigtable service shared by many users. LRU assigns 3.4x–3.5x as much flash to workload 1, and obtains 4.5x–5.1x as high a flash read rate as FIFO. This accounts for a 14.4–16.7 percentage-point increase in the cell-wide flash hit rate. Even optimized FIFO does not achieve this high a flash read rate for workload 1, because the workload's cacheability function is much steeper for LRU than for FIFO.

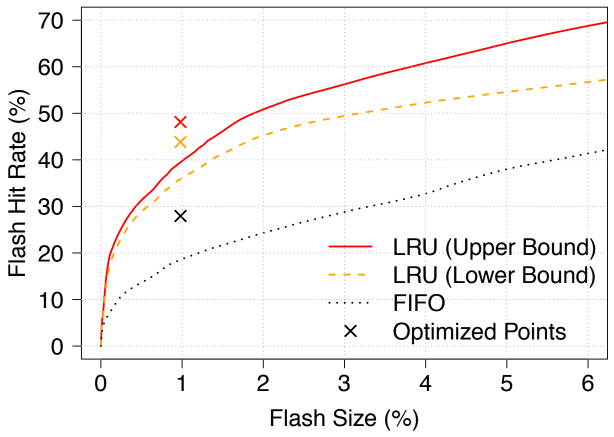

Figure 10: Cacheability curve for a single FIFO and single LRU. The marked points represent optimized flash hit rates for FIFO and LRU.

Normalized Flash Hit Rate (%)

Workload	FIFO	Opt FIFO	LRU	Opt LRU
1	4.1	5.3	18.5 – 20.8	15.8 – 16.9
2	0.0	7.3	0.0 – 0.0	7.3 – 7.3
3	0.1	0.9	1.3 – 2.1	1.6 – 6.1
4	3.0	1.9	5.6 – 6.1	4.9 – 5.3
5	6.4	5.7	4.6 – 4.7	6.8 – 5.2
Others	4.9	6.7	5.8 – 5.9	7.2 – 7.2
Total	18.5	27.8	35.9 – 39.5	43.6 – 47.9

Normalized Flash Size Percentage (%)

Workload	FIFO	Opt FIFO	LRU	Opt LRU
1	0.13	0.22	0.46 – 0.44	0.20 – 0.15
2	0.00	0.08	0.00 – 0.00	0.08 – 0.08
3	0.00	0.08	0.03 – 0.03	0.04 – 0.17
4	0.18	0.03	0.11 – 0.11	0.07 – 0.08
5	0.42	0.35	0.20 – 0.21	0.38 – 0.28
Others	0.25	0.22	0.18 – 0.19	0.22 – 0.21
Total	0.98	0.98	0.98 – 0.98	0.98 – 0.98

Table 5: Flash hit rate and size per workload assuming single and optimized FIFO/LRU. Workloads are ordered in decreasing order of flash read rate under Optimized LRU. The two numbers for LRU respectively use the lower and upper bounds of the cacheability function. Assumes cell-wide flash size percentage of 1% during the training period, which became 0.98% during evaluation since the amount of data increased slightly.

Table 5 also shows that workload 2 accounts for most of the difference between optimized and single versions of FIFO/LRU. In fact, both optimized FIFO and LRU put the entire contents of workload 2 on flash, increasing the cell-wide flash read hit by 7.3%. Workload 2 is a Bigtable used to serve static webpage content.

These results are robust to adjusting the period used for training. Of our 7-day dataset, we used the 7th day for evaluation and the 6th day for training; the remaining days were used only to compute the file ages. If the 5th day is used instead for training, then the cell-wide flash hit rate is 28.5% under optimized FIFO, and 44.4–49.1% under optimized LRU. Using the 4th day, we get 27.0% under optimized FIFO and 42.7–48.1% under optimized LRU. These numbers are similar to those in Table 5.

While LRU eviction performs better than FIFO for many workloads, there is a substantial associated overhead. The LRU age of a file depends on accesses to all its component chunks, and hence the eviction scanner must gather information from multiple chunkservers before determining whether a file should be evicted. By comparison, computing the FIFO age is simple because it depends only on the static creation time of the file.

9 Conclusions

The falling price of flash storage has made it cost-effective for some workloads to fit entirely in flash. As the I/O rate per byte supported by disks continues to decline, flash storage also becomes a critical component of the storage mix for many more workloads in modern storage systems. However, because flash is still expensive, it is best to use it only for workloads that can make good use of it. With Janus, we show how to use long-term workload characterization to determine how much flash storage should be allocated to each workload in a cloud-scale distributed file system.

Janus builds a compact representation of the cacheability of different user I/O workloads based on sampled RPC traces of I/O activity. These *cacheability curves* for different users are used to construct a linear optimization problem to determine the flash allocations that maximize the read hits from flash, subject to operator-set priorities and write-rate bounds.

This system has been in use at Google for 6 months. It allows users to make informed flash provisioning decisions by providing them a customized dashboard showing how many reads would be served from flash for a given flash allocation. Another view helps system administrators make allocation decisions based on a fixed amount of flash available in order to maximize the reads offloaded from disk.

Based on evaluations from workloads using these recommendations and I/O traces of other workloads, we conclude that the recommendation system is quite effective. In our trace-based estimates, flash hit rates using the optimized recommendations are 47-76% higher than the option of using the flash as an unpartitioned tier. We find that application owners appreciate learning how much flash is cost-effective for their workload.

Acknowledgments

Janus would not have been possible without the help of many individuals and teams. We are especially grateful to Jun Luo, Adam Gee, Denis Serenyi, and the entire Colossus team for their early collaboration on this project. Andy Chu, Herb Derby, Lawrence Greenfield, Sean Quinlan, Paul Cychosz, Salim Virji, and Gang Ren also contributed ideas or helped with the data collection and analysis on which Janus is built. We are grateful to John Wilkes, Florentina Popovici, our shepherd Kai Shen, and our anonymous referees for their feedback on improving the presentation.

References

[1] Retrieved 2013/01/09: `http://www.google.com/shopping/product/7417866799343902880/specs`.

[2] AGUILERA, M. K., ET AL. Improving recoverability in multi-tier storage systems. In *DSN* (2007), IEEE, pp. 677–686.

[3] ALVAREZ, G. A., ET AL. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst. 19*, 4 (2001), 483–518.

[4] ANDERSON, E., ET AL. Quickly finding near-optimal storage designs. *ACM Trans. Comput. Syst. 23*, 4 (2005), 337–374.

[5] BAKER, M., ET AL. Measurements of a distributed file system. In *SOSP* (1991), ACM, pp. 198–212.

[6] BLAZE, M. A. *Caching in large-scale distributed file systems*. PhD thesis, Princeton University, 1993.

[7] CANAN, D., ET AL. *Using ADSM Hierarchical Storage Management*. IBM Redbooks. 1996.

[8] CHANG, F., ET AL. Bigtable: a distributed storage system for structured data. In *OSDI* (2006), USENIX, pp. 205–218.

[9] COEHLO, N., MERCHANT, A., AND STOKELY, M. Uncertainty in aggregate estimates from sampled distributed traces. In *Workshop on Managing Systems Automatically and Dynamically (MAD 2012)*, USENIX.

[10] GASIOR, G. SSD prices down 38% in 2012, but up in Q4, 2013. Retrieved 2013/01/29: http://techreport.com/review/24216/ssd-prices-down-38-in-2012-but-up-in-q4.

[11] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *SOSP* (2003), ACM, pp. 29–43.

[12] GRAY, J., AND PUTZOLU, F. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. In *SIGMOD* (1987), ACM, pp. 395–398.

[13] GUERRA, J., ET AL. Cost effective storage using extent based dynamic tiering. In *FAST* (2011), USENIX, pp. 273–286.

[14] KROEGER, T., AND LONG, D. Design and implementation of a predictive file prefetching algorithm. In *ATC* (2001), USENIX, pp. 105–118.

[15] LOBOZ, C. Z. Cloud resource usage: extreme distributions invalidating traditional capacity planning models. In *Workshop on Scientific Cloud Computing (ScienceCloud 2011)*, ACM, pp. 7–14.

[16] MASSIGLIA, P. Exploiting multi-tier file storage effectively. Retrieved 2013/01/29: `https://snia.org/sites/default/education/tutorials/2009/spring/file/PaulMassiglia_Exploiting_Multi-Tier_File_StorageV05.pdf`, 2009.

[17] MCKUSICK, M. K., AND QUINLAN, S. GFS: Evolution on fast-forward. *Communications of the ACM 53*, 3 (2010), 42–49.

[18] NARAYANAN, D., ET AL. Migrating server storage to SSDs: analysis of tradeoffs. In *EuroSys* (2009), ACM, pp. 145–158.

[19] OH, Y., ET AL. Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage filesystems. In *FAST* (2012), USENIX.

[20] OUSTERHOUT, J. K., ET AL. A trace-driven analysis of the UNIX 4.2 BSD file system. In *SOSP* (1985), ACM, pp. 15–24.

[21] PATTERSON, R. H., ET AL. Informed prefetching and caching. In *SOSP* (1995), ACM, pp. 79–95.

[22] SIGELMAN, B. H., ET AL. Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Google, Inc., 2010.

[23] STOKELY, M., ET AL. Projecting disk usage based on historical trends in a cloud environment. In *Workshop on Scientific Cloud Computing (ScienceCloud 2012)*, ACM, pp. 63–70.

[24] WILKES, J., GOLDING, R. A., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst. 14*, 1 (1996), 108–136.