Flamingo: Enabling Evolvable HDD-based Near-Line Storage

Sergey Legtchenko
Microsoft Research

Xiaozhou Li[*]
Microsoft Research

Antony Rowstron
Microsoft Research

Austin Donnelly
Microsoft Research

Richard Black
Microsoft Research

Abstract

Cloud providers and companies running large-scale data centers offer near-line, cold, and archival data storage, which trade access latency and throughput performance for cost. These often require physical rack-scale storage designs, e.g. Facebook/Open Compute Project (OCP) Cold Storage or Pelican, which co-design the hardware, mechanics, power, cooling and software to minimize costs to support the desired workload. A consequence is that the rack resources are restricted, requiring a software stack that can operate within the provided resources. The co-design makes it hard to understand the end-to-end performance impact of relatively small physical design changes and, worse, the software stacks are brittle to these changes.

Flamingo supports the design of near-line HDD-based storage racks for cloud services. It requires a physical rack design, a set of resource constraints, and some target performance characteristics. Using these Flamingo is able to automatically parameterize a generic storage stack to allow it to operate on the physical rack. It is also able to efficiently explore the performance impact of varying the rack resources. It incorporates key principles learned from the design and deployment of cold storage systems. We demonstrate that Flamingo can rapidly reduce the time taken to design custom racks to support near-line storage.

1 Introduction

Storage tiering has been used to minimize storage costs. The cloud is no exception, and cloud providers are creating near-line cloud storage services optimized to support cold or archival data, for example Amazon's Glacier Service [2], Facebook's Cold Data Storage [17], Google near-line storage [19] and Microsoft's Pelican [8]. In contrast to online storage [16], near-line storage trades data access latency and throughput for lower cost; access latencies of multiple seconds to minutes are normal and throughput is often lower or restricted.

To achieve the cost savings many of these near-line storage services use custom rack-scale storage designs, with resources such as power, cooling, network bandwidth, CPU, memory and disks provisioned appropriately for the expected workload. This is achieved through co-designing the rack hardware and software together, and systems like Pelican [8] and OCP Cold Storage [20] have publicly demonstrated that designing custom racks for near-line storage can result in significant cost savings. For example, in both of these designs there is insufficient rack-level power provisioned to allow all the hard disk drives (HDDs) to be concurrently spinning. By implication, the rack cooling is then only provisioned to handle the heat generated from a subset of the HDDs spinning. The goal of the rack's storage stack is to achieve the best possible performance without exceeding the physical resource provisioning in the rack.

The most common way of managing these constrained resources is by controlling how data is striped across HDDs, and by ensuring the individual IO requests are scheduled taking into account resource provisioning. In particular, the *data layout* defines the set of disks for a single IO that need to be read from or written to, and the *IO scheduler* defines the set of disks that need to be accessed for multiple IOs being concurrently performed. Our experience building near-line storage is that, given a well-designed storage stack, it is feasible to only redesign the data layout and IO scheduler in the stack to handle different rack designs and/or performance goals. Unfortunately, it is also the case that even simple and seemingly small design changes *require* a redesign of the data layout and IO scheduler. Designing the data layout and IO scheduler is challenging and time consuming even for experts, and it is hard to know if they are achieving the best possible performance from the rack.

Flamingo is a system that we use to help automate and

[*]Currently a PhD. student at Princeton.

reduce the complexity of designing near-line storage. It incorporates the many lessons learned during the design and deployment of Pelican. Flamingo uses a generalized storage stack that is derived from the one used in Pelican and described in [8], and a tool chain to automatically synthesize the configuration parameters for the storage stack. Flamingo requires a physical rack description, a set of resource descriptions in the form of resource constraints, and expected performance characteristics. Under typical operation the tool chain takes a few hours to produce the configuration parameters. Flamingo has been used to determine the impact of and to drive design and component changes to Pelican.

Flamingo is also able to help designers explore the physical rack design space by automatically quantifying the impact of varying the physical resource provisioning in the rack. It is able to determine the minimum increase in a resource, such as power, that would yield a change in performance. It is also able to determine the impact of using components with different properties, such as a new HDD with a different power profile. In such cases, it can also evaluate how much extra performance could be gained by reconfiguring the storage stack to exploit that component. Flamingo can handle significantly more complexity than a human and it is able to generate configurations and determine the likely performance of a physical design before it is even built.

This paper is organized as follows: Section 2 introduces near-line storage, Pelican and motivates the problems solved by Flamingo. Section 3 and 4 describe Flamingo, and the core algorithms used. Section 5 shows results, Section 6 describes related work and Section 7 concludes.

2 Background: Near-line storage

A cloud-scale storage service will consist of thousands of storage racks. A deployed rack will be used for many years, and then retired. Rack designs will be revised as price points for components change or newer versions are released. Hence, at any point in time, a small number of different storage rack designs will be deployed in a single cloud-scale storage service. A near-line storage rack will usually consist of servers and HDDs, and each server will run an instance of a storage stack. In online storage it is common to have 30-60 HDDs per server, while in near-line it can be 500+ HDDs per server. We provide a brief overview of Pelican as Flamingo uses many of its key principles, but for the full details see [8].

Pelican A Pelican rack has 1,152 HDDs and two servers. Each HDD is connected to a SATA 4-port multiplier, which is connected to a 4-port SATA HBA. Pelican uses PCIe to connect the 72 HBAs to the server, such that each HBA can be attached to either one of the servers. Power and cooling are provisioned to allow only a small fraction of the HDDs to concurrently be spinning and ready to perform IO (active) while the other HDD platters are spun down (standby).

HDDs are physically located in multiple physical resource domains: power, cooling, vibration and bandwidth. A Pelican power domain contains 16 HDDs and has sufficient power to support two HDDs transitioning from standby to active, with the 14 other HDDs in standby. A Pelican cooling domain has 12 HDDs and can provide sufficient heat dissipation to support one HDD transitioning from standby to active and 11 in standby. These domains represent constraints imposed by the physical rack, and combining these two constraints means that at most 96 HDDs can be concurrently active in a Pelican.

Violating physical resource constraints leads to transient failures, can increase hardware failure rates or simply decrease performance. Hence, the storage stack needs to ensure that the operating state of the rack remains within provisioned resources. Pelican handles these constraints by first carefully managing data layout. Each HDD is assigned to a group that contains 24 HDDs. The assignment is done to ensure all HDDs in a group can be concurrently transitioned from standby to active. Hence, at most 2 HDDs per group can be in the same power domain. Pelican stripes a stored file across multiple HDDs in the same group and, if required, erasure coding can be used. The Pelican prototype striped a file across eighteen HDDs with fifteen data fragments and three redundancy fragments. The mapping of HDDs to groups, the group and stripe size and erasure coding parameters are the *data layout configuration*. They are a function of number of HDDs in the rack, the physical resource constraints, required data durability, target throughput, and the capacity overhead. They are unique to a particular hardware design and set of resource constraints. To determine them is complex and during the original Pelican design it took many months to determine the correct parameters.

Within the Pelican software stack the other part which interacts closely with the physical rack and resource constraints is the IO scheduler. The IO scheduler determines the order in which IO requests are serviced, and it attempts to balance performance with fairness. Flamingo uses a new IO scheduler that is configurable and we discuss this in detail in Section 3.2.

Real-world lessons Pelican makes a number of simplifying assumptions. Notably, it assumes that an active HDD uses the same resources as a HDD transitioning from standby to active. This makes the problem more tractable, but can lead to resource underutilization that results in lower performance than theoretically supported. Some elements of the Pelican software stack

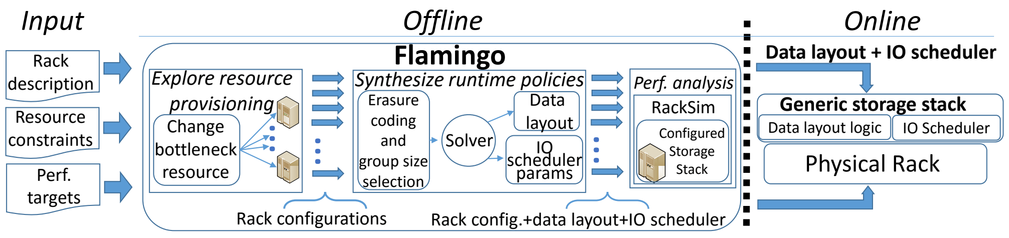


Figure 1: Flamingo overview.

proved to be very brittle to design changes. Subtle and often seemingly innocuous changes to the physical rack or components require significant redesign of the data layout and IO scheduler. For example, during the design of Pelican changing HDDs introduced new vibration issues, and also changed the power and cooling profiles. These changes provided the potential to have more HDDs to be concurrently active. However, without redesigning the data layout and IO scheduling in Pelican we were unable to unlock the better performance these HDDs could offer. This also requires using HDDs that offer similar or better properties compared to the HDDs we designed for originally. Subtle changes can result in resources being violated, which is often hard to detect when they do not lead to immediate failure. Finally, the cooling properties of a rack are a function of the ambient temperature in the data center in which it operates. This varies across data center designs and data center cooling technologies. This means that to maximize resource usage given a set of physical resources then a per data center data layout and IO scheduler is required.

When building complex near-line storage systems it is hard to accurately estimate the performance impact of small design changes. Simplistically, adding a few extra fans will increase the fraction of HDDs that can be concurrently active in a cooling domain, but it is hard to understand the impact this will have on higher-level performance metrics.

Finally, we also believe that, based on our experiences with Pelican for near-line storage, the underlying principle in Pelican of organizing the HDDs into groups that can be migrated between states concurrently is good. This allows resource conflicts to then be handled at the group level rather than the HDD level which lowers complexity and increases performance. The design of Flamingo therefore embodies this concept.

3 Flamingo

Flamingo leverages the fact that most of the Pelican storage stack is generic and independent of the hardware constraints; it uses the Pelican stack with a new configurable IO scheduler, and then uses offline tools to synthesize the data layout and IO scheduler parameterization for a given rack design.

Flamingo also supports the co-design of the rack hardware and software. Using a rack-scale event-based simulator it allows potential hardware resource configurations to be instantiated with a storage stack and then specific workloads replayed against them to understand higher-level performance. It also automatically explores the design space for resource provisioning to determine the performance impact of increasing the resources in a rack. This information can be used to both change the physical rack design, but also to help component manufacturers optimize their components to yield better performance.

Figure 1 shows the two main Flamingo components: an offline tool and a configurable storage stack. The offline tool has three phases. The first takes a physical rack description and a set of resource constraints, and iteratively generates new sets of resource constraints that effectively provide the potential for higher performance. The physical rack description and a single set of resource constraints represents a potential configuration, and requires a parameterized storage stack. The second phase then concurrently synthesizes for each unique configuration the parameters required for the data layout and the online IO scheduler. Target performance characteristics are provided, and the goal is to synthesize the configuration for the storage stack that meets or exceeds these performance characteristics. If it can be determined in this phase that a particular performance target cannot be met, then a policy can be specified to either relax the performance target or simply reject the configuration.

Unfortunately, not all performance targets can be verified as being met or exceeded during the second phase, and the final phase uses an accurate rack-scale discrete event simulator to empirically determine the expected performance. This does a parameter sweep using synthetic and real workloads evaluating micro- and macro-level performance for each configuration point. At the end of this offline process Flamingo has generated the storage stack parameters for each configuration, and the relative performance of each configuration. If the exploration of multiple configurations is not required, then the first stage can be skipped.

Failure Domains
$failure_{rate} : [comp_{name}, AFR]$
$failure_{domain} : [dom_{name}, comp_{name}, \{HDD_{id1}, HDD_{id2}, ..., HDD_{idN}\}]$

Resource Domains
$HDD_{cost} : [res_{name}, [cost_{standby}, cost_{spinningup}, cost_{spinning}]]$
$resource_{domain} : [dom_{name}, res_{name}, \{HDD_{id1}, HDD_{id2}, ..., HDD_{idN}\}, dom_{budget}, hard\|soft]$

Figure 2: The rack description and resource constraints.

Flamingo is able to perform this in less than 24 hours for all rack configurations that we have tried. We now describe in detail how Flamingo works, starting with the information Flamingo requires.

3.1 Flamingo requirements

Flamingo requires a rack description that captures the different resources and their domains, a set of resource constraints expressed over these domains, the target performance characteristics and a state machine that captures the HDD operating states and resource usage. The rack description captures the physical properties of the rack and the physical relationship between HDDs and resource domains. The set of resource constraints capture the resource provisioning per resource domain. Given the tight relationship between the rack description and the resource constraints we use a single input file to capture them both, and its syntax is shown in Figure 2.

Each HDD is assigned a unique HDD_{id}. Flamingo allows an arbitrary number of HDD operating states. For simplicity here, we use only three states: standby, spinning up and active. In reality, there are several other potential states, including multiple lower power states, some of which keeps the platter spinning at a low RPM. Flamingo requires a state machine showing the possible HDD operating states and transitions between them. Operating states where IO requests can be serviced need to be explicitly identified, e.g. when the HDD is not spinning, or is spinning up, the HDD is unable to service IO requests. Operating states that can service IO requests are referred to as active. Given the current design of HDDs the tool supports only a single active state currently. Flamingo also needs to know which operating states are transient, e.g. spinning up.

The rack description and the resource constraints are expressed in terms of sets of HDDs. The rack description includes information about all (component) failure domains with their associated Annualized Failure Rates (AFR). If the AFR varies over time, then the worst case AFR is specified. Each *failure domain* is expressed in terms of the set of HDDs that would become inaccessible if the component fails. For example, if there is a tray that connects 16 HDDs together, a tray failure will lead to all 16 HDDs failing. So, if there are b trays then there will be b sets each containing 16 HDDs and an AFR will be associated with the tray.

The resource constraints are captured as *resource domains* which are expressed as a set of HDDs and an associated resource budget. Examples of resource domains may be power, cooling, bandwidth, and vibration. Individual HDDs will appear in multiple resource domains. Flamingo uses no explicit knowledge of any resource types, it treats all resources as simply names with associated constraints. This allows new resources to be easily incorporated within a design or arbitrarily changed. For example, half way through the design of Pelican we realized that the layout needed to handle vibration. Because Flamingo has no knowledge of resource types, a budget is associated with each resource domain set, and is simply a floating point number, and the unit is arbitrary. For example, for a power resource domain the unit could be Watts, and the original budget could be 50W. For each resource, Flamingo also needs the resource cost for operating in each state (HDD_{cost}), in the case of power these can be taken from the data sheet, e.g. spinning up may be 20W, active may be 10W and standby may be 1W. The *current cost* is the sum for all HDDs for them to operate in their current operating state. If a resource domain is *hard* then the current cost must not be higher than the budget, as this can cause long or short term failure. A $soft$ resource domain can be violated, but this will impact performance rather than failure rates. For each resource domain it is possible to set an upper bound that is used to control the search space when exploring changing the resource provisioning. By default, when exploring the design space Flamingo will look to increase a resource by the minimum that will allow at least one drive to transition to a different operating state. The minimum increase can also be specified. For example, a power domain may have an upper bound of 500W and a minimum increase of 25W.

Hierarchical resource domains can easily be expressed. For example, there could be a backplane that has 10 trays with 16 HDDs attached to it. A power domain can be created containing all 160 HDDs with a power budget. Then a power domain can also be created for each of the 10 trays. The sum of the tray budgets can exceed the budget for the backplane, but the backplane budget will never be exceeded.

Some resources are not necessarily additive, for example vibration. Using resource domains and budgets we have been able to handle these by emulating counting semaphores. The budget is used to capture the number of HDDs that are allowed in a particular state, and the HDD costs are set to zero or one. Using overlapping resource domains then also allows us to specify complex relationships. One set of resource constraints could be used to enforce that no neighboring HDDs can spin up concurrently, while a second one says that in a single tray only

4 can spin up concurrently. Flamingo will enforce both in its designs.

Finally, Flamingo also requires target performance characteristics; in particular data durability, physical servicing window, rack deployment lifetime, lower bound on bandwidth for file transfer, level of IO concurrency, capacity overhead for failure resilience and a fairness goal expressed as the trade-off in access latency versus throughout.

Simplifying assumptions The rack description allows arbitrary racks to be described. However, Flamingo makes two assumptions about the resource domains. First, for each resource defined *every* HDD in the rack must be specified in a $resource_{domain}$ description for that resource. For example, if power is a resource then each HDD must appear in at least one $resource_{power}$ definition. Second, each resource domain definition for a resource must include the same number of HDDs and be provisioned with the same budget. In the previous example of a tray and backplane power domain with different number of HDDs, this can be simply encoded by naming the resource domains differently, e.g. $power_{tray}$ and $power_{backplane}$. Finally, we assume that there is only one class of storage device specified. Flamingo can support other classes of storage device beyond HDD, provided they can be expressed as having multiple operating states over different resources. Flamingo could be extended to handle different storage device classes in the same rack, but this would increase the state space that Flamingo needs to handle. We believe these assumptions are reasonable and hold for all cold storage hardware that we are aware of, including Pelican and OCP Cold Storage. They simplify the data layout and in many cases reduce the number of inter-group constraints, improving concurrency and reducing overhead for the IO scheduler.

3.2 Flamingo Design

We now describe three important aspects of the core Flamingo design: the exploration of rack configurations, the data layout and the IO scheduler configuration.

3.2.1 Exploring rack configurations

The offline tool has three phases, the first explores the design space for resource provisioning in the rack. This is achieved by taking a configuration consisting of the rack description and a set of resource constraints and slowly relaxing the resource constraints. Each time a resource constraint is relaxed a new configuration is created which consist of the original rack description with the new set of resource constraints.

The intuition is that, if there are q resources, then there is a large q-dimensional space representing the set of all configurations. However, many of these configurations will vary resources that are not impacting the performance and can therefore be ignored. Hence, there is a surface being defined in the q-dimensional space of interesting configurations that can impact performance, and Flamingo is determining the configurations that lie on that surface. This can be a large space, for example a simple Pelican has $q = 4$ and, given multiple operating states for the HDDs, the total number of potential configurations is in the millions. However, the number of useful configurations will be considerably smaller.

Flamingo achieves this by determining the bottleneck resource for a given configuration. To calculate the bottleneck resource Flamingo calculates the number of HDDs in the rack (N) and, for each hard resource r, Flamingo determines the number of HDDs in each resource domain set for r, (N_r), and the per-resource domain budget (r_{budget}). Both N_r and r_{budget} will be the same for all resource domain sets for r. We define $cost_{highest}$ as the highest cost HDD operating state and the lowest as $cost_{lowest}$. The number of HDDs, (m_r), that can be in the highest operating state in each single resource domain is:

$$m_r = \left\lfloor \frac{r_{budget} - cost_{lowest} N_r}{cost_{highest} - cost_{lowest}} \right\rfloor \tag{1}$$

Across the entire rack the number of HDDs, (M_r), that can be operating in their highest cost operating state for the resource is:

$$M_r = (N/N_r) \times m_r \tag{2}$$

Flamingo generates for each resource r the value M_r. Given two resources, say $r = power$ and $r = cooling$, then *power* is more *restrictive* than *cooling* if $M_{power} < M_{cooling}$. To determine the bottleneck resource, the resources are ordered from most to least restrictive using their M_r values. The most restrictive resource is the *bottleneck resource*. The maximum number of HDDs that can be concurrently in their highest cost operating state M is then simply $M = M_{bottleneckresource}$. If there are two or more resources with equal M_r values then it is recorded that there are multiple bottleneck resources.

Once a bottleneck resource has been identified, the budget associated with the bottleneck resource is increased by δ. δ is the maximum of the smallest additional cost that will allow a single HDD in the bottleneck resource domain to transition to the next highest cost operating state and the specified minimum increase for the resource domain. The budget is then increased on the bottleneck domain by δ to create a new configuration.

If there is more than one bottleneck resource, then a new configuration is created where exactly one resource is selected to be relaxed. These configurations are then all used independently to recursively generate more configurations. The configuration exploration terminates when $M = N$, in other words, represents a fully provisioned rack or the bottleneck resource has reached the upper bound specified for it and cannot be increased. If the bottleneck resource cannot be increased it does not matter if other resources could be increased, they cannot yield better performance.

The number of configurations considered is dependent on the number of resources and the range over which the resources operate. Generating the configurations is fast, taking on the order of seconds on a high end CPU. Once all the configurations have been generated the storage stack parameters need to be calculated, which can happen in parallel for each configuration.

3.2.2 Data Layout

For each configuration Flamingo next needs to synthesize the data layout, and this involves two stages:

Groups and erasure coding Flamingo computes *groups* of HDDs such that each HDD belongs to a single group and there are sufficient resources across all resource domains to allow all the HDDs in the group to concurrently transition to their active state. We make a simplifying assumption that all groups are the same size, n. A file is stored on a subset of the HDDs in a single group, with k data and r redundant fragments generated for each file using erasure coding [28, 12].

The first stage is to calculate the group size. Flamingo does this by initially generating a set of candidate group sizes. Files are stored in a single group, therefore n should be large enough to store all fragments of a file even in presence of HDD failures, but small enough to maximize the number of groups that can be concurrently spun up. Because all HDDs in a group need to be able to spin up concurrently, $\lfloor M/n \rfloor$ groups can be simultaneously activated. To maximize resource utilization, we first enforce that $M \bmod n = 0$. For example, if $M = 96$ then both $n = 21$ and $n = 24$ allow the same number of concurrently active groups: 4, but only $n = 24$ fulfills $96 \bmod n = 0$. For $M = 96$, this restricts the possible group sizes to $n = \{1, 2, 3, 4, 8, 12, 16, 24, 32, 48, 96\}$. We refer to this as the *candidate set*. If the set is empty, then Flamingo stops processing the configuration and generates an error.

Flamingo then determines a set of values for erasure coding parameters k and r. The choice of values are a function of *(i)* the required data durability, *(ii)* the component failure rates, *(iii)* the storage capacity redundancy overhead *i.e.,* $\frac{r}{k+r}$, *(iv)* the interval between physically servicing a rack, and *(v)* the lower bound on per-file read or write throughput. The first four parameters are used in a simple failure model to generate a set of possible $k + r$ values. The fifth parameter is then used as a threshold for values of $k + r$, removing combinations that would yield too low throughput, so we look for $k \times HDD_{bandwidth} \geq target$. The result is an ordered list consisting of $k + r$ pairs that provide the specified durability ranked by the storage capacity overhead ($\frac{r}{k+r}$). If the set is empty, then an error is raised and Flamingo stops processing this configuration. The same model is also used to calculate f, an estimate of the maximum number of HDDs expected to fail during a rack service interval. This is calculated assuming that failure recovery is performed at the rack level which can be done by the Flamingo storage stack. However, if failure recovery is handled at a higher level across storage racks, then f can be configured to always be zero.

Given the candidate set of possible group sizes, the ranked $(k + r)$ list and f, Flamingo needs to select the lowest value for n from the candidate set, such that $k + r + f \leq n$. This maximizes the number of concurrently active groups and therefore the number of concurrent IO requests that can be serviced in parallel. So, given the previous candidate groups sizes, if the smallest value of $(k, r) = (15, 3)$ and $f = 2$ then $n = 24$ will be selected. If M/n is less than the specified concurrent IO request target, Flamingo stops processing the configuration.

The Flamingo storage stack attempts to distribute the stored data in a group uniformly across all the HDDs in a group. When a group is accessed all n HDDs are concurrently migrated to the new state, rather than k. The reason to spin up $k+r$ is to allow us to read the data when the first k HDDs are ready to be accessed. The Flamingo runtime spins up the entire n (e.g. $k + r + f$) HDDs opportunistically, because if another request arrives for the group we are able to service it without waiting for potentially another drive to spin up.

Mapping HDDs to groups Once n has been determined, Flamingo next needs to form l, where $l = N/n$, groups and assign each HDD to exactly one group. The assignment is static, and transitioning *any* HDD in a group to a new state that would violate any hard resource constraint means the entire group cannot transition.

The assignment must also try to maximize IO request concurrency, which means maximizing the number of groups that can concurrently transition into active, where the upper bound is M/n. However, ensuring a mapping that achieves this is non-trivial because each HDD assigned to a group potentially conflicts with other groups in all its domains. This will lead to inefficient data layouts, in which every group conflicts with $l - 1$ groups, achieving very low IO request concurrency e.g. one.

The number of possible assignments grows exponentially with the number of HDDs. To make this tractable, we use a custom designed solver that restricts the search space and selects the best group assignment according to a set of performance-related characteristics and heuristics. The solver exploits the observation that many resource domains are not composed of arbitrary HDDs but are rather defined by their physical location in the rack. For instance, the power domain would correspond to a backplane. The solver derives a coordinate system that captures this physical layout from the rack description and assigns a d-dimensional coordinate to each HDD, where d is the number of resource domain types.

The solver tries to form groups of HDDs that are close to each other in the coordinate space and do not conflict in any resource domain. It does this by initially generating different ordered vectors of the HDDs. This is achieved by changing the starting coordinate and ranking the coordinates on different dimensions. Hence, if each HDD has an (x,y) coordinate, one ranking would be generated by ordering on x then y and another one would be generated ranking y then x. The ordering function is dimension specific, so it can generate smallest to largest on x, but for coordinates where x is equal, rank largest to smallest on y. This generates multiple orderings of the HDDs. For each ordered vector created Flamingo greedily attempts to assign HDDs to groups, using a number of different heuristics to control into which group the next HDD is mapped. This is deterministic, no randomization is used. Intuitively, this finds good assignments because the group structure exploits the physical symmetry of the rack topology, forming sets of groups that conflict in all domains and are independent from the rest of the rack.

For each iteration, if the solver finds a solution where all HDDs are successfully assigned to groups such that all the HDDs in each group can concurrently transition operating states, then Flamingo needs to measure the quality of each solution. The metric of importance is the level of IO request concurrency that can be achieved by the data layout. An efficient solution will always allow any arbitrary selected M/n groups to be concurrently in their highest operating state.

Even with the custom solver this metric will need to be calculated potentially thousands of times per configuration. Hence, Flamingo uses a number of fast-to-compute heuristics. First, Flamingo determines if the groups are symmetric. We take each resource constraint and replace the HDD identifier in the definitions with the group identifier. For each group we then look at each resource domain in which it is present, and count the number of other unique groups that are present in each. We refer to these groups as conflicting groups. If, across all groups, the cardinality of the conflicting groups is the same, then the groups are symmetric. Each group impacts the same number of other groups. Further, the expected upper bound on the number of groups that should conflict with each group can be calculated.

Flamingo then uses a sub-sampling of the space to check configurations, and in particular explores sample sets consisting of less than or equal to M/l groups, checking if they can be successfully concurrently transitioned. The sub-sampling also estimates a lower bound on the number of groups that can be active (e.g. spinning) and another group transitioned into an active state. The expected number is determined as a function of M and again sub-sampling is used to estimate the lower bound. The number of samples can be varied *per configuration*.

If the ranking is shown to have no examples that violate the expected performance for these heuristics, then it is marked *efficient* and the solver stops. Otherwise, the solver records the quality of the metrics and continues to iterate through rankings. If all rankings have been checked and no efficient solutions found, then the solver selects the best solution found but marks the result *inefficient*. The output of the solver is a set of HDD to group mappings which define the data layout.

3.2.3 IO scheduler

Once data layout is complete the IO scheduler configuration needs generating. The IO scheduler in the storage stack receives IO requests and controls the order in which they are executed. It also controls when groups transition states. If it has a request for a group that is currently not active, it will ensure that the group becomes active and then issues the request to be serviced. It has to ensure that during operation the order in which groups transition between states does not violate the resource constraints. In order to do this, the IO scheduler needs to understand the relationship between groups, and we achieve this using a set of constraints between groups. The inter-group constraints capture the resource sharing relationships between groups, and allow the IO scheduler to determine which groups can concurrently be spinning.

To generate these IO scheduler *group constraints* Flamingo translates the resource constraints from being HDD based to group based. Each HDD identifier in each resource constraint is replaced with the HDD's group identifier and a weight, w_{id} initially set to one. For each resource constraint, all references to same group identifier are combined into a single entry with w_{id} being set to the number of references. The budget and associated per state costs for the original resource constraints are kept. If there are multiple group constraints which have *exactly* the same groups represented, the one with the most restrictive budget is kept. Flamingo outputs the set of group constraints.

The online IO scheduler in the storage stack uses the group constraints to control which groups can be spun up. It maintains a per-group queue for IO requests that are yet to be issued and an operating state for each group,

Rack	#HDDs	#HDDs/server	#domains	avg. HDDs/domain
OCP	240	240	73	15
Pelican	1152	576	1111	10
Rack_A	1152	576	1039	22
Rack_B	1152	576	1087	11
Rack_C	1152	576	1063	14
Rack_D	960	480	942	9
Rack_E	1920	960	1883	9

Table 1: Core properties of the seed racks.

which maps onto the HDD states, e.g. standby, spinning up, and spinning. The IO scheduler also maintains for each group constraint a balance, equal to the sum of $cost_{state} \times w_{id}$ for each group. In general, a group can transition to a new state if, for all group constraints, the change in balance is within the group constraint budget.

The IO scheduler is invoked each time a IO request is received, or an IO or group state transition completes. It needs to determine if there is a request that can be now serviced or if a group transition needs to occur to service queued requests.

The choice of which group or groups to transition is a function of the per group queue depth and the current queuing delay for the head request. There is a trade-off between latency and throughput, there is a throughput penalty for changing group state, but there is a latency penalty of making requests queue for longer. The performance characteristics specified control this trade-off. If the IO scheduler decides that a group g needs to transition state, the IO scheduler iterates over the groups and, using the group constraints, greedily identifies sets of groups that could be transitioned to free the resources to allow g to transition. If none or insufficient resources are found, then the scheduler waits for in-flight requests or group transitions to complete. If there are a number of sets of groups, then the scheduler selects the groups to transition based on their queue depth and head request delay. When it has selected a group or groups to transition, if there are spare resources in any group constraints, the IO scheduler is invoked again to allow further groups to transition state.

4 Evaluation

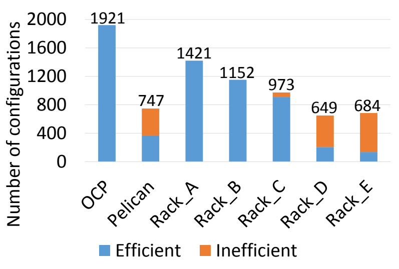
We now evaluate the performance of Flamingo using seven seed rack configurations, including Pelican and the Facebook/OCP Cold Storage Design [20]. The OCP Cold Storage Rack contains two independent servers and 16 Open Vault chassis, each filled with two trays of 15 HDDs with sufficient power and cooling to support one active drive and 14 in standby. The tray is a vibration domain, and each server is connected to 8 chassis using SAS containing a combined 240 HDDs and independent of the other server in the rack. Hence, this rack configuration is a half rack consisting of a server and 240 HDDs. Details of the software stack have not been released, but a Pelican-like storage stack is needed as most HDDs will be in standby. The other five racks are based on other cold storage designs and we refer to them as Rack_A to Rack_E. Table 1 summarizes the number of HDDs, the number of resource domains and the average HDDs per resource domain for each of them. All the designs have multiple bandwidth resource domains, to capture the bandwidth from the HDDs to the server, as well as power, cooling and vibration domains. Racks A to E are all credible physical hardware design points for cold storage which vary the power, cooling, and HDD density (hence vibration and HDD-to-server bandwidth). We have built out Pelican and Rack_D. We put no upper bounds or increment limits on the resource domains for any resources in any rack.

Flamingo uses a rack-scale discrete event simulator to estimate the performance of rack with the synthesized data layout and IO scheduler. The simulator is based on the discrete event simulator used to evaluate Pelican, which we have extended to support arbitrary physical rack topologies and to use the constraint-aware IO scheduler described. It models HDDs, network bandwidth and the server-to-HDD interconnect, and is configured with mount, unmount and spin up latency distributions from measurements of real archive class HDDs and has been cross validated against real rack-scale storage designs (for example the prototype Pelican [8]).

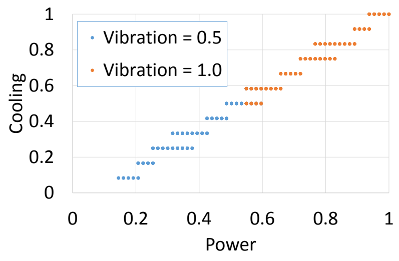
In the experiments we used a cluster of servers, each with two Intel Xeon E5-2665 2.4Ghz processors and 128 GB of DRAM. For each configuration we do a parameter sweep over a range of possible workload characteristics. A sequence of client read requests for 1 GB files is generated using a Poisson process with an average arrival rate $\lambda = 0.0625$ to 5. Beyond $\lambda = 5$ the network bandwidth becomes the bottleneck for all racks. The read requests are randomly distributed across all the files stored in the rack. We simulate 24 hours, and gather statistics for the last 12 hours when the simulation has reached a steady state. We believe this workload allows comprehensive comparison of the rack configurations.

4.1 Flamingo performance

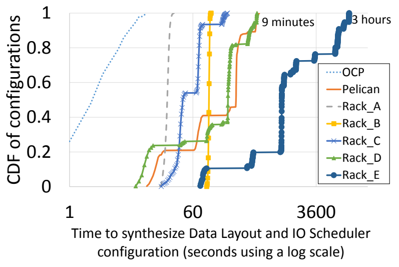
First we evaluate the performance of Flamingo exploring the resource design space and creating the configurations from the initial rack description. For each of the seven racks, the time to generate the derived configurations is less than three seconds on a single server. Figure 3(a) shows the total number of configurations derived for each rack. Across each of the racks there is wide variance in the number of configurations derived, 649 to 1,921. The



(a) Number of configurations.



(b) Configurations for Pelican.



(c) Execution time.

Figure 3: Base performance of Flamingo.

number of configurations is a function of the resource domains and which ones are the bottleneck resource in each configuration. Across all racks 7,547 configurations are created.

Figure 3(a) also shows for each rack the fraction of configurations for which the generated data layout was considered efficient or inefficient by Flamingo. If Flamingo finds a configuration in which *(i)* all HDDs are allocated to a group, and *(ii)* all HDDs in a single group can be migrated from standby to active concurrently, it uses the fast heuristics as described to determine if the solution is efficient or inefficient. If these two conditions do not hold then the configuration is marked as having no solution, however for all 7,547 configurations a layout (efficient or inefficient) was found.

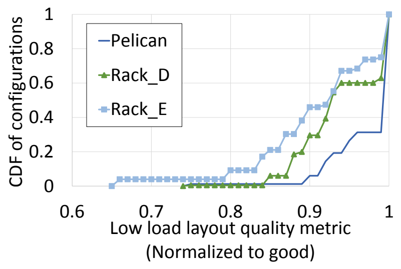
Figure 3(b) shows the fraction of power, cooling and vibration provisioned in each configuration derived from the Pelican rack. Each point represents a configuration and power and cooling are shown on the two axes, normalized to being fully provisioned. Hence a value of (1,1) means that the resource is sufficiently provisioned to have all HDDs in the rack in their most resource-consuming operating state. The vibration domain is expressed using the color of the point, again normalized to fully provisioned. Although showing only three resources, Figure 3(b) demonstrates how Flamingo traverses the design space, incrementing the bottleneck resource each time. For each configuration we increment the bottleneck resource by the smallest unit that will allow a single HDD to be in a more expensive operating state. However, this does not necessarily mean that the bottleneck resource changes from the previous configuration. In Figure 3(b) the impact of this can be seen where there are multiple power configurations for each step in the cooling.

Execution time Next, we consider the execution time of Flamingo. The solver used to synthesize the data layout and IO scheduler for each configuration runs as an independent single threaded process for each configuration. Flamingo runs one instance of the solver on each core of each server it is run on. Figure 3(c) shows a CDF of derived racks versus time taken to generate the data layout and the IO scheduler configuration for each configuration. The time taken is a function of the complexity of the configuration and the number of HDDs, and for all except those for Rack_E, none takes more than 9 minutes. In the worst case, for a configuration derived from Rack_E it takes 3 hours and the median for this rack is 20 minutes. The time taken for Flamingo is heavily dominated by the number of HDDs; as the number of HDDs increases the size of the state space to search increases faster then linearly. Table 1 shows Rack_E has 1,920 HDDs, almost a factor of two larger than the other racks. Our solver is deterministic and can report as it executes both the current best found solution and the fraction of the search space it has explored.

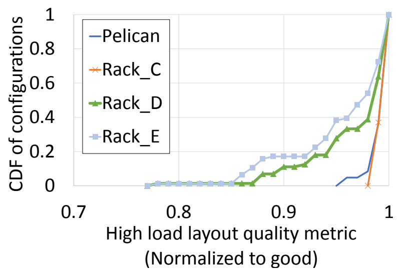
Once the data layout and IO scheduler parameters have been created, Flamingo runs the simulator to estimate the performance of each configuration. The time taken by the simulator is a function of the workloads evaluated. The workloads used in this paper allow a comprehensive exploration of the relative performance and across all 7,547 configurations we observed a mean execution time of 45 minutes per configuration, with a maximum of 83 minutes. As with the parameter generation, the simulations can be run concurrently.

4.2 Data layout quality

Next we quantify the quality of the data layout generated for each configuration. Flamingo considers a layout as efficient or inefficient, and stops searching once it finds one it considers efficient. Analytically it is impossible to determine if a layout is optimal at these scales, so instead we use two metrics. The first metric is the number of groups that can be concurrently spun up, which is a good indicator of performance under low load. For a configuration we can determine the bottleneck resource, and using that we can calculate an upper bound on the number of groups that should be able to be concurrently active in their highest state (m). We then generate a ran-



(a) Low load metric.

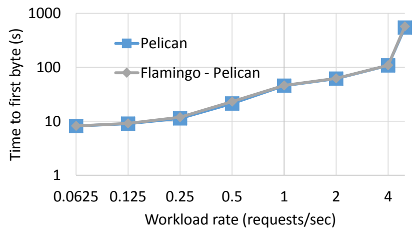


(b) High load metric.

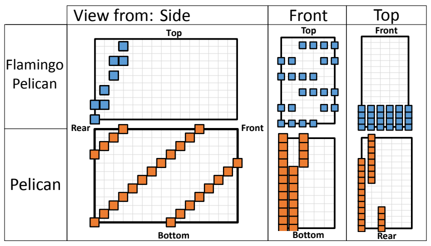
Figure 4: Quality of data layout.

Rack	Configurations	Efficient	False Positive	False Negative
OCP	1921	1921	0	0
Pelican	747	369	0	0
Rack_A	1421	1421	0	0
Rack_B	1152	1152	0	0
Rack_C	973	909	361	0
Rack_D	649	205	0	9
Rack_E	684	135	39	39

Table 2: Quality of Flamingo's data layout heuristics.



(a) Performance.



(b) Group layout.

Figure 5: Flamingo vs Pelican.

dom test ordering of the n groups in the configuration. For each configuration we greedily try to spin up the first k groups, where $k = 1, 2, 3, \ldots, m$. If we are able to migrate the HDDs in the k groups from standby to active concurrently without violating any resource constraints, we remove the group at position k in the test ordering and try again. Eventually, there are no further groups to remove from the test ordering and $k < m$, or $k = m$. We repeat this 250,000 times ensuring a unique ordering for each trial and record k and normalize it to m. We refer to this as the low load quality metric and reflects the level of concurrency achievable under low load.

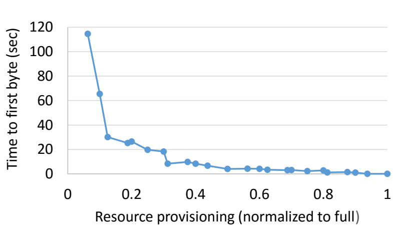
The second metric is the number of groups that can be concurrently active and still allow an additional group to become active. This is a good indicator of performance under high load. We use the same process to calculate this metric, except instead of concurrently migrating the HDDs in all group from standby to active, we leave $k-1$ active and try to transition the kth group to the spinning up state. Again, we can calculate the value of m for this metric using the bottleneck resource. We refer to this as the high load quality metric. If, for all 250,000 trials, both metrics are one then the data layout is considered *good* otherwise it is considered *bad*. These metrics are not used by the Flamingo solver as they take many hours to compute for each *single* solution, and need to be computed for all the large number of solutions considered.

Table 2 compares using these metrics to the fast heuristics used by Flamingo showing the total number of configurations, the number of these configurations that Flamingo said it could generate an efficient layout, and then the number of false positives and false negatives. A *false positive* is a configuration marked efficient by Flamingo but bad by the metrics. A *false negative* is marked inefficient by Flamingo but good by the metrics. Three racks: OCP, Rack_A and Rack_B, have efficient and good layouts for all configurations.

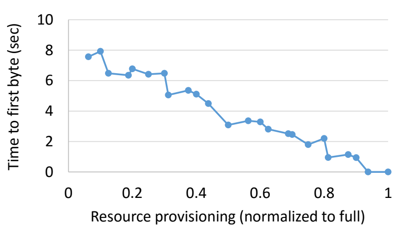
In order to understand further the quality of inefficient configurations, as well as the false positives and negatives, Figure 4 shows a CDF of configurations versus both quality metrics when the metrics are not one (OCP, Rack_A and Rack_B omitted). The low load metric is not 1 for only three racks, and in all cases the median is above 0.9. Under the high load metric all solutions are at 0.75 or higher for the four racks. This shows that even when a rack is not efficient, the quality of the solutions is high.

4.3 Storage performance

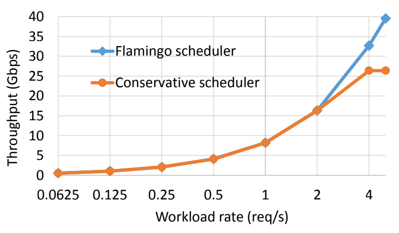
The last set of experiments evaluate the rack performance when using the IO scheduler and the data layout synthesized by Flamingo. First we compare how Flamingo performs to a manually-designed solution. To do this we



(a) Latency, high workload rate - Rack_E



(b) Latency, low workload rate - Rack_E



(c) Throughput - OCP rack

Figure 6: Performance of generated layouts and IO scheduler.

take the rack configuration which is equivalent to the Pelican rack used in [8] and compare its performance with the Flamingo generated data layout and IO scheduler constraints.

We first consider the time to first byte, which is the time between a request being issued by a client and the first data byte being sent to the client. This includes all queuing and spin up delays. Figure 5(a) shows the time to first byte as a function of the workload rate. Flamingo-Pelican is using the data layout and IO scheduler configuration synthesized by Flamingo and Pelican is the manually designed layout and IO scheduler. The time to first byte for Pelican and Flamingo-Pelican is virtually identical across all workload rates. This is true for other properties measured by the simulator, including throughput, which are omitted due to space constraints.

The fact that Flamingo-Pelican matches the original performance is interesting. The data layout differs significantly from the original Pelican layout. Figure 5(b) shows a representation of the rack as a 3D array of HDDs shown from the front, side and top comparing the layout of group zero for each. Each colored square represents a HDD, and other groups can be thought of as rotations of this group. Flamingo groups cover different individual resource domains compared to the original Pelican.

The next experiment explores the ability of Flamingo to exploit the resources provisioned in each rack. Ideally, the performance should increase as the provisioning of the bottleneck resource increases. For each of the 7,547 configurations for the seven racks we measure the time to first byte as a function of resource provisioning. Figure 6(a) shows the time to first byte versus the resource provisioning. Due to lack of space, we use a single (high) workload rate of 4 requests per second and only show results for one randomly selected rack (Rack_E). The other racks show similar trends in the results. To quantify the resource provisioning we use its M value normalized by the total number of HDDs in the rack representing a fully provisioned rack. Recall that M is the maximum number of HDDs that can be concurrently in their highest cost operating state, and is a function of the bottleneck resource. While deriving configurations, Flamingo increases the bottleneck resource budget by a value δ which is potentially less than the cost of allowing a HDD to operate in the highest cost operating state, hence several configurations can share the same M value.

From Figure 6(a) we see that the time to first byte generally decreases as provisioning increases, meaning that Flamingo is able to adapt to the increased resource provisioning, achieving better performance with more resources. The performance improvement is not monotonic: in some cases, the resource provisioning increase does not decrease the time to first byte. This happens because Flamingo attempts to optimize for general performance across multiple metrics, rather than just time to first byte. Figure 6(a) also shows that the decrease in time to first byte is not linear as the provisioning is increased. When resources are scarce, even a slight increase in provisioning leads to significantly better performance. For example, increasing the provisioning from 0.06 to 0.1 leads to a time to first byte decreased by nearly 80% on average for Rack_E. We observe this trend for all seven racks, meaning relatively low provisioned racks can achieve a performance close to fully provisioned ones. Intuitively, this happens because for the given workload, resource provisioning within the rack is not necessarily the bottleneck. At some point, the performance becomes limited by external factors such as the bandwidth from the rack to the data center fabric (in this case 40 Gbps). Notably, the exact benefit of increasing resources is very different for each initial rack description, e.g. for Rack_A, the time to first byte decreases by 80% only when resource provisioning reaches 0.68.

To illustrate this further we use a low workload rate of 0.0625 requests per second. Figure 6(b) shows the time to first byte versus the resource provisioning for Rack_E. For this low workload rate, the IO scheduler is unable to do extensive batching of requests and needs to frequently transition between groups. The rack bandwidth is not the bottleneck and the IO scheduler can benefit from more resources in the rack to increase concurrency of group transitioning. As a result, the time to first byte decreases almost linearly as provisioning increases. Resource provisioning depends on multiple factors internal *and* external to the rack. Tools like Flamingo provide great benefit when co-designing a rack and storage stack for a particular workload.

The final experiment evaluates the benefit for the IO scheduler to dynamically manage the available resources. Pelican made the simplifying assumption that HDDs could have two states; standby and active. This leaves some resources unused which means that it will be able to keep fewer groups concurrently active, but has the benefit of being much simpler and we refer to this as a conservative IO scheduler. Allowing an arbitrary number of states with differentiated costs requires the IO scheduler to track transitions between each state for all HDDs, and ensuring that budgets will not be violated by each transition to a new state. We compare the conservative and the Flamingo schedulers using the OCP rack. For this default configuration power is the bottleneck resource, with sufficient provisioning to allow two groups to spin up concurrently. Figure 6(c) shows the throughput as a function of the workload rate. For workloads with higher request rates of 2 or more requests/second, the Flamingo IO scheduler outperforms the conservative one. It does this because, at the higher loads, it can keep more groups concurrently spinning; it is able to keep up to three groups concurrently spinning as opposed to two for the conservative scheduler, allowing one more requests to be processed in parallel. For lower workload rates, the performance is dominated by the number of groups that can spin up concurrently as the IO scheduler needs to frequently transition between groups, so the Flamingo IO scheduler offers no additional performance. It should be noted that if the HDDs can operate in lower power RPM states which offer faster transitioning to active, the benefit of the finer-grained resource management in the Flamingo IO scheduler would enable increased performance for all workload rates.

5 Related Work

Flamingo addresses the challenges of designing rack-scale systems for near-line storage. To reduce costs physical resources are typically constrained. The storage stack needs to maximize performance without violating the constraints making data layout and IO scheduling key. In contrast, traditional storage is provisioned for peak performance. There have been proposals for systems like MAID [10], as well as other power efficient storage systems [22, 18, 4, 24, 32], that allow idle disks to spin down. Data layout and mechanisms to handle spun down disks is important in all their designs. Pergamum [22] used NVRAM to handle meta-data and other small writes, effectively providing a write-back cache used when the disks are spun down. Hibernator [32] supports low RPM disk modes and dynamically determines the proportion of disks in each mode in function of the workload. Rabbit [4], Sierra [24] and PARAID [27] achieve power-proportionality through careful data layout schemes, but in these systems fine-grained provisioning of physical resources is not done at design time.

There has been work on automatic configuration of RAID storage [21, 30, 29, 3, 1, 6, 7], for example to design RAID configuration that meet workload availability requirements [3, 1, 6, 9]. These use a solver that takes declarative specifications of workload requirements and device capabilities, formulates constraint representation of each design problem, and uses optimization techniques to explore the search space of possible solutions computing the best RAID level for each logical unit of data on disk. Designs often include an online data migration policy between RAID levels [30, 7]. Flamingo is designed to optimize the physical resource utilization in the rack, working at a larger scale and explicitly handling a large number of constrained resources.

Tools to manage the design and administration of enterprise [26, 5], cluster [15] and wide-area [13] storage that optimize for data availability, durability and capital cost as primary metrics offline but do not consider fine-grained resource management or online IO scheduling.

Flamingo provides quantitative answers to questions about hypothetical workload or resource changes and their impact on performance. This is similar to prior work [25, 23, 11]. For example, [23] evaluates different storage provisioning schemes, which helps understanding trade-offs. In contrast, Flamingo complements the analysis by creating the data layout and IO scheduling policies for each configuration.

More generally, [14] proposes automatically generating data layout for data-parallel languages. Remy [31], given network characteristics and transport protocol targets, synthesizes a network congestion control algorithm. Flamingo has the same high-level goal: to make systems less brittle.

6 Conclusion

Flamingo is designed to simplify the development of rack-scale near-line storage. Flamingo has two high-level goals: first to synthesize the data layout and IO scheduler parameters for a generic storage stack for cloud near-line storage racks. The second aspect is that Flamingo supports the co-design of rack hardware and software, by allowing an efficient exploration of the impact of varying the resources provisioned within the rack.

Acknowledgments We would like to thank Aaron Ogus, the reviewers and our shepherd Mustafa Uysal for their feedback.

References

[1] ALVAREZ, G. A., BOROWSKY, E., GO, S., ROMER, T. H., BECKER-SZENDY, R., GOLDING, R., MERCHANT, A., SPASOJEVIC, M., VEITCH, A., AND WILKES, J. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst. 19*, 4 (Nov. 2001), 483–518.

[2] Amazon glacier. `http://aws.amazon.com/glacier/`, August 2012.

[3] AMIRI, K., AND WILKES, J. Automatic Design of Storage Systems To Meet Availability Requirements. Tech. Rep. HPL-SSP-96-17, Computer Systems Laboratory, Hewlett-Packard Laboratories, August 1996.

[4] AMUR, H., CIPAR, J., GUPTA, V., GANGER, G. R., KOZUCH, M. A., AND SCHWAN, K. Robust and Flexible Power-proportional Storage. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), SoCC '10.

[5] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. C. Hippodrome: Running circles around storage administration. In *FAST* (2002), vol. 2, pp. 175–188.

[6] ANDERSON, E., SPENCE, S., SWAMINATHAN, R., KALLAHALLA, M., AND WANG, Q. Quickly Finding Near-optimal Storage Designs. *ACM Trans. Comput. Syst. 23*, 4 (Nov. 2005), 337–374.

[7] ANDERSON, E., SWAMINATHAN, R., VEITCH, A. C., ALVAREZ, G. A., AND WILKES, J. Selecting raid levels for disk arrays. In *FAST* (2002), vol. 2, Citeseer, pp. 189–201.

[8] BALAKRISHNAN, S., BLACK, R., DONNELLY, A., ENGLAND, P., GLASS, A., HARPER, D., LEGTCHENKO, S., OGUS, A., PETERSON, E., AND ROWSTRON, A. Pelican: A Building Block for Exascale Cold Data Storage. In *OSDI* (Oct. 2014).

[9] BOROWSKY, E., GOLDING, R., MERCHANT, A., SCHREIER, L., SHRIVER, E., SPASOJEVIC, M., AND WILKES, J. Using attribute-managed storage to achieve qos. In *Building QoS into distributed systems*. Springer, 1997, pp. 203–206.

[10] COLARELLI, D., AND GRUNWALD, D. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing* (2002), SC '02, IEEE Computer Society Press, pp. 1–11.

[11] EL MALEK, M. A., II, W. V. C., CRANOR, C., GANGER, G. R., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M., PRASAD, M., SALMON, O., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., WACHS, M., AND WYLIE, J. J. Early Experiences on the Journey Towards Self-* Storage. *IEEE Data Eng. Bulletin 29* (2006).

[12] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., YEKHANIN, S., ET AL. Erasure coding in windows azure storage.

[13] KEETON, K., SANTOS, C. A., BEYER, D., CHASE, J. S., AND WILKES, J. Designing for disasters. In *FAST* (2004), vol. 4, pp. 59–62.

[14] KENNEDY, K., AND KREMER, U. Automatic Data Layout for Distributed-memory Machines. *ACM Trans. Program. Lang. Syst. 20*, 4 (July 1998), 869–916.

[15] MADHYASTHA, H. V., MCCULLOUGH, J., PORTER, G., KAPOOR, R., SAVAGE, S., SNOEREN, A. C., AND VAHDAT, A. scc: cluster storage provisioning informed by application characteristics and slas. In *FAST* (2012), p. 23.

[16] MARCH, A. Storage pod 4.0: Direct wire drives - faster, simpler, and less expensive. `http://blog.backblaze.com/2014/03/19/backblaze-storage-pod-4/`, March 2014.

[17] MORGAN, T. P. Facebook loads up innovative cold storage datacenter. `http://tinyurl.com/mtc95ve`, October 2013.

[18] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write off-loading: Practical power management for enterprise storage. *Trans. Storage 4*, 3 (Nov. 2008), 10:1–10:23.

[19] NEWSON, P. Whitepaper: Google cloud storage nearline. `https://cloud.google.com/files/GoogleCloudStorageNearline.pdf`, March 2015.

[20] OPEN COMPUTE STORAGE. `http://www.opencompute.org/projects/storage/`.

[21] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1988), SIGMOD '88, ACM.

[22] Storer, M. W., Greenan, K. M., Miller, E. L., and Voruganti, K. Pergamum: Replacing Tape with Energy Efficient, Reliable, Disk-based Archival Storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), FAST'08.

[23] Strunk, J. D., Thereska, E., Faloutsos, C., and Ganger, G. R. Using utility to provision storage systems. In *FAST* (2008), vol. 8, pp. 1–16.

[24] Thereska, E., Donnelly, A., and Narayanan, D. Sierra: Practical power-proportionality for data center storage. In *Proceedings of the Sixth Conference on Computer Systems* (2011), EuroSys '11.

[25] Thereska, E., Narayanan, D., and Ganger, G. R. Towards Self-predicting Systems: What if You Could Ask "What-if"? *Knowl. Eng. Rev. 21*, 3 (Sept. 2006), 261–267.

[26] Ward, J., O'Sullivan, M., Shahoumian, T., and Wilkes, J. Appia: Automatic storage area network fabric design. In *FAST* (2002), vol. 2, p. 15.

[27] Weddle, C., Oldham, M., Qian, J., Wang, A.-I. A., Reiher, P., and Kuenning, G. Paraid: A gear-shifting power-aware raid. *ACM Transactions on Storage (TOS) 3*, 3 (2007), 13.

[28] Wicker, S. B., and Bhargava, V. K. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[29] Wilkes, J. Traveling to Rome: A Retrospective on the Journey. *SIGOPS Oper. Syst. Rev. 43*, 1 (Jan. 2009), 10–15.

[30] Wilkes, J., Golding, R., Staelin, C., and Sullivan, T. The HP AutoRAID Hierarchical Storage System. *ACM Trans. Comput. Syst. 14*, 1 (Feb. 1996), 108–136.

[31] Winstein, K., and Balakrishnan, H. TCP Ex Machina: Computer-generated Congestion Control. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (2013), SIGCOMM '13.

[32] Zhu, Q., Chen, Z., Tan, L., Zhou, Y., Keeton, K., and Wilkes, J. Hibernator: helping disk arrays sleep through the winter. In *ACM SIGOPS Operating Systems Review* (2005), vol. 39, ACM, pp. 177–190.