Pelican: A building block for exascale cold data storage

Shobana Balakrishnan, Microsoft Research
Richard Black, Microsoft Research
Austin Donnelly, Microsoft Research
Paul England, Microsoft Research
Adam Glass, Microsoft Research
Dave Harper, Microsoft Research
Sergey Legtchenko, Microsoft Research
Aaron Ogus, Microsoft
Eric Peterson, Microsoft Research
Antony Rowstron, Microsoft Research

Abstract

A significant fraction of data stored in cloud storage is rarely accessed. This data is referred to as *cold data*; cost-effective storage for cold data has become a challenge for cloud providers. Pelican is a rack-scale hard-disk based storage unit designed as the basic building block for exabyte scale storage for cold data. In Pelican, server, power, cooling and interconnect bandwidth resources are provisioned by design to support cold data workloads; this *right-provisioning* significantly reduces Pelican's total cost of ownership compared to traditional disk-based storage.

Resource right-provisioning in Pelican means only 8% of the drives can be concurrently spinning. This introduces complex resource management to be handled by the Pelican storage stack. Resource restrictions are expressed as constraints over the hard drives. The data layout and IO scheduling ensures that these constraints are not violated. We evaluate the performance of a prototype Pelican, and compare against a traditional resource over-provisioned storage rack using a cross-validated simulator. We show that compared to this over-provisioned storage rack Pelican performs well for cold workloads, providing high throughput with acceptable latency.

1 Introduction

Cloud storage providers are experiencing an exponential growth in storage demand. A key characteristic of much of the data is that it is rarely read. This data is commonly referred to as being *cold data*. Storing data that is read once a year or less frequently in online hard disk based storage, such as Amazon S3, is expensive, as the hardware is provisioned for low latency access to the data [7]. This has led to a push in industry to understand and build out cloud-scale tiers optimized for storing cold data, for example Amazon Glacier [1] and Facebook Cold Data Storage [8]. These systems attempt to minimize up front capital costs of buying the storage, as well as the costs of running the storage.

In this paper we describe Pelican, a prototype rack-scale storage unit that forms a basic building block for building out exabyte-scale cold storage for the cloud. Pelican is a converged design, with the mechanical, hardware and storage software stack being co-designed. This allows the entire rack's resources to be carefully balanced, with the goal of supporting only a cold data workload. We have designed Pelican to target a peak sustainable read rate of 1 GB per second per PB of storage (1 GB/PB/sec), assuming a Pelican stores 1 GB blobs which are read in their entirety. We believe that this is higher than the actual rate required to support a cold data workload. The current design provides over 5 PB of storage in a single rack, but at the rate of 1 GB/PB/s the entire contents of a Pelican could be transferred out every 13 days.

All aspects of the design of Pelican are *right-provisioned* to the expected workload. Pelican uses a 52U standard-sized rack. It uses two servers connected using dual 10 Gbps Ethernet ports to the data center network, providing an aggregate of 40 Gbps of full-duplex bandwidth. It has 1,152 archive-grade hard disks packed into the rack, and using currently available drives of average size 4.55 TB provides over 5PB of storage. Assuming a goodput of approximately 100 MB/s for a hard disk drive, including redundancy overheads, then only 50 active disks are required to sustain 40 Gbps.

A traditional storage rack would be provisioned for peak performance, with sufficient power and cooling to allow all drives to be concurrently spinning and active. In Pelican there is sufficient cooling to allow only 96 drives to be spun up. All disks which are not spun up are in standby mode, with the drive electronics powered but the platters stationary. Likewise we have sufficient power for only 144 active spinning drives. The PCIe-bus is stretched out across the entire rack, and we provision it to have 64 Gbps of bandwidth at the root of the

PCIe-bus, much less than 1 Gbps per drive. This careful provisioning reduces the hardware required in the rack and increases efficiency of hardware layout within the rack. This increases storage density; the entire rack drive density is significantly higher than any other design we know, and reduces peak and average power consumption. All these factors result in a hardware platform significantly cheaper to build and operate.

This *right-provisioning* of the hardware, yet providing sufficient resources to satisfy the workloads, differentiates us from prior work. Massive Arrays of Idle Disks (MAID) systems [5] and storage systems that achieve power-proportionality [15] assume that there is sufficient power and cooling to have all disks spinning and active when required. Therefore, they simply provide a power saving during operation but still require the hardware and power to be provisioned for the peak. For a Pelican rack the peak power is approximately 3.7 kW, and average power is around 2.6 kW. Hence, a Pelican provides both a lower capital cost per disk as well as a lower running cost, whereas the prior systems provide just a lower running cost.

However, to benefit from our hardware design we need a software storage stack that is able to handle the restrictions of having only 8% of the disks spinning concurrently. This means that Pelican operates in a regime where spin up latency is the modern day equivalent of disk seek latency. We need to design the Pelican storage stack to handle the disk spin up latencies that are on the order of 10 seconds.

The contributions of this paper are that we describe the core algorithms of the Pelican software stack that give Pelican good performance even with hardware restricted resources, in particular how we handle data layout and IO scheduling. These are important to optimize in order to achieve high throughput and low per operation latency. Naive approaches yield very poor performance but carefully designing these algorithms allows us to achieve high-throughput with acceptable latency.

To support data layout and IO scheduling Pelican uses groups of disks that can be considered as a schedulable unit, meaning that all disks in the group can be spun up concurrently without violating any hardware restrictions. Each resource restriction, such as power, cooling, vibration, PCIe-bandwidth and failure domains, is expressed as constraints over sets of physical disks. Disks are then placed into one of 48 groups, ensuring that all the constraints are maintained. Files are stored within a single group and, as a consequence, the IO scheduler needs to schedule in terms of 48 groups rather than 1,152 disks. This allows the stack to handle the complexity of the right-provisioning.

We have built out a prototype Pelican rack, and we present experimental results using that rack. In order to allow us to compare to an over-provisioned storage rack we use a rack-scale simulator to compare the performance. We cross-validate the simulator against the full Pelican rack, and show that it is accurate. The results show that we are able to sustain a good throughput and control latency of access for workloads up to 1 GB/PB/sec.

The rest of this paper is organized as follows. Section 2 provides an overview of the Pelican hardware, and describes the data layout and IO Scheduler. Section 3 describes a number of issues we discovered when using the prototype hardware, including issues around sequencing the power-up with hardware restrictions. In Section 4 we evaluate the performance of Pelican and the design choices. Related work is detailed in Section 5. Finally, Section 6 discusses future work and Section 7 concludes.

2 A Pelican Rack

A Pelican stores unstructured, immutable chunks of data called *blobs* and has a key-value store interface with `write`, `read` and `delete` operations. Blobs in the size range of 200 MB to 1 TB are supported, and each blob is uniquely identified by a 20 byte key supplied with the operation.

Pelican is designed to store blobs which are infrequently accessed. Under normal operation, we assume that for the first few months blobs will be written to a Pelican, until a target capacity utilization is hit, and from then on blobs will be rarely read or deleted during the rest of the lifetime of the Pelican until it is decommissioned. Hence the normal mode of operation for the lifetime of the rack will be servicing reads, and generating internal repair traffic when disks fail or are replaced.

We also assume that Pelican is used as a lower tier in a cloud-based storage system. Data is staged in a higher tier awaiting transfer to a Pelican, and the actual time when the data is migrated is under the control of the target Pelican. This means that writes can always occur during quiescent or low load periods. Therefore, we focus on the performance of a Pelican for read-dominated workloads.

We start by providing an overview of the Pelican hardware, before describing in detail how the storage stack performs data placement and request scheduling to handle the right-provisioned hardware.

2.1 Pelican hardware

Pelican is a 52U rack filled with 1,152 archival class 3.5” SATA disks. Pelican uses a new class of archival drive manufactured for cold storage systems. The disks are placed in trays of 16 disks. The rack contains six 8U chassis each containing 12 trays (192 disks) organized

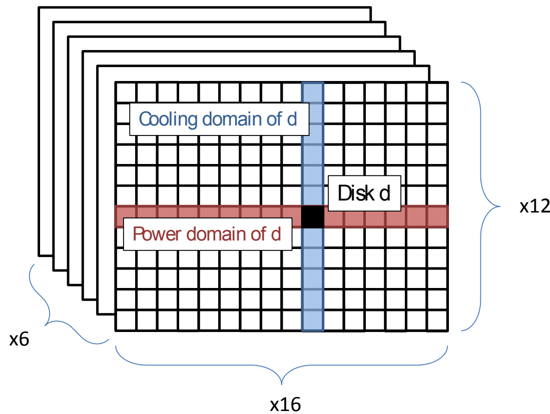


Figure 1: Schematic representation of the Pelican rack

in 2 rows of 6 trays. Each tray is approximately 4U high and the disks are inserted as 8 pairs back-to-back and vertically mounted in the tray. Each chassis has a backplane connecting the 12 horizontally laid trays. Figure 1 shows a schematic representation of a Pelican, a cuboid of 6 (width) x 16 (depth) x 12 (height) disks where the power is shared by disks in the same row and cooling shared by disks in the same column. The backplane supplies power to the 72 trays, and this is currently configured to be sufficient to support two active spinning or spinning up disks. A tray is an independent *power domain*. Due to the nature of power distribution, and the use of electronic fuses, we provision the system to support a symmetrical power draw across all trays. In-rack cooling uses multiple forced air flow channels, each cooling a section of multiple trays, the *cooling domain*. Each air flow is shared by 12 disks. There are 96 independent cooling domains in the rack, and each is currently calibrated to support cooling only one active or spinning up disk. Pelican supports the cooling of 96 drives, but there is sufficient power to have 144 drives spinning. The number of cooling domains and trays is in part driven by convenience for the mechanics and physical layout. Given that approximately 50 drives need to be active in order to allow a Pelican to generate 40Gbps of network traffic, having 96 drives active at any one time, allows us to be servicing requests concurrently with spinning up drives for queued requests.

The SATA disks in the trays are connected to a Host Bus Adapter (HBA) on the tray. The tray HBAs connect to a PCIe switch on the backplane. This supports *virtual switching*, allowing the physical switch to be partitioned into two virtual switches, each with a root port connected to one of the two servers. Each HBA is connected to one of the two virtual switches; this selection is dynamic and under software control.

Each of the six chassis are connected to both servers (12 cables in total) and each PCIe hierarchy provides only 64 Gbps bandwidth at its root. Under normal operation the rack is vertically partitioned into two halves (of 36 trays) with no shared cooling or power constraints. Each server can therefore independently and concurrently handle reads and writes to its half of the rack. However, if a server fails, the PCIe virtual switches can be reconfigured so all disks attach to a single server. Hence, servers, chassis, trays and disks are the *failure domains* of the system.

This right-provisioning of power, cooling and internal bandwidth resources permits more disks per rack and reduces total cost of ownership. The cost reduction is considerable. However, the storage software stack must ensure that the individual resource demands do not exceed the limits set by the hardware.

2.2 Pelican Software Storage Stack

The software storage stack has to minimize the impact on performance of the hardware resource restrictions. Before describing the data layout and IO scheduling algorithms used in Pelican we more formally define the different resource domains.

2.2.1 Resource domains

We assume that each disk uses resources from a set of *resource domains*. Resource domains capture right-provisioning: a domain is only provisioned to supply its resource to a subset of the disks simultaneously. Resource domains operate at the unit of disks, and a single disk will be in multiple resource domains (exactly one for each resource).

Disks that are in the same resource domain for any resource are *domain-conflicting*. Two disks that share no common resource domains are *domain-disjoint*. Disks that are domain-disjoint can move between disk states *independently*, and it is guaranteed there will be no over-committing of any resources. Disks which are domain-conflicting cannot move independently between states, as doing so could lead to resource over-commitment. Most normal storage systems provisioned for peak performance only take into account only failure domains. With resource right-provisioning we increase considerably the number of domains and so the complexity.

In order to allow the scheduling and the placement to be efficient, we express resource domains as constraints over the disks. We classify the constraints as *hard* or *soft*. Violating a hard constraint causes either short- or long-term failure of the hardware. For example, power and cooling are hard constraints. Violating the power constraint causes an electronic fuse to trip that means drives are not cleanly unmounted, and potentially dam-

aging the hardware. A tray becomes unavailable until the electronic fuse (automatically) resets. Violating the cooling constraint bakes the drives, reducing their lifetime. In contrast, violating a soft constraint does not lead to failure but instead causes performance degradation or inefficient resource usage. For example, bandwidth is a soft constraint: the PCIe bus is a tree topology, and violating the bandwidth constraint simply results in link congestion resulting in a throughput drop.

The Pelican storage stack is designed to enforce operation within the hardware constraints with performance. The Pelican storage stack uses the following constraints: *(i)* one disk spinning or spinning up per cooling domain; *(ii)* two disks spinning or spinning up per power domain; *(iii)* shared links in the PCIe interconnect hierarchy; and *(iv)* disks located back-to-back in a tray share a vibration domain. Like other storage systems we also have failure domains, in our case for disks, trays and chassis.

2.2.2 Data layout

In order to provide resiliency to failures, each blob is stored over a set of disks selected by the data layout algorithm. Blob placement is key as it impacts the concurrency of access to blobs.

When a blob is to be written into a Pelican, it is split into a sequence of 128 kB data fragments. For each k fragments we generate r additional fragments containing redundancy information using a Cauchy Reed-Solomon erasure code [4] and store the $k+r$ fragments. We use a systematic code, which means if the k original fragments are read back then the input can be regenerated by simply concatenating these fragments[1]. Alternatively, a reconstruction read can be performed by reading *any* k fragments, allowing the reconstruction of the k input fragments. We refer to the $k+r$ fragments as a *stripe*. To protect against failures, we store a stripe's fragments on independent disks. For efficiency all the fragments associated with a single blob are stored on the same set of $k+r$ disks. Further, all the fragments of the blob on a single disk are stored contiguously in a single file, called a *stripe stack*. While any k stripe stacks are sufficient to regenerate a blob, the current policy is to read all the $k+r$ stripe stacks on each request for the blob to allow each stripe stack to be integrity checked to prevent data corruption. Minimally we would like to ensure that the $k+r$ drives that store a blob can be concurrently spun up and accessed.

In the current Pelican prototype we are using $k = 15$ and $r = 3$. First, this provides good data durability with the anticipated disk and tray annual failure rates (AFRs) with a reasonable capacity overhead of 20%. The $r = 3$ allows up to three concurrent disk failures without data loss. Secondly, during a single blob read Pelican would like saturate a 10 Gbps network link, and reading from 15 disks concurrently can be done at approximately 1,500 MB/s or 12 Gbps provided that all the disks are spun-up and are chosen to respect the PCI bandwidth constraints. Thirdly, it is important that each blob is mapped to 18 domain-disjoint disks. If the disks cannot be spinning concurrently to stream out the data then in-memory buffering at the servers proportional to the size of the blob being accessed would be needed. If the disks are domain-disjoint then at most $k+r$ 128 kB fragments need to be buffered in memory for each read, possibly reconstructed, and then sent to the client. Similarly for writes only the fragments of the next stripe in the sequence are buffered in memory, the redundancy fragments are generated, and all fragments sent to disk.

The objective of the data layout algorithm is to maximize the number of requests that can be concurrently serviced while operating within the constraints. Intuitively, there is a queue of incoming requests, and each request has a set of disks that need to be spinning to service the request. We would like to maximize the probability that, given one executing request, we can find a queued request which can be serviced concurrently.

To understand how we achieve layout we first extend the definition of domain-conflict and domain-disjointness to sets of disks as follows: two sets, s_a and s_b are domain-conflicting if *any* disk in s_a is domain conflicted with *any* disk in s_b. Operations on domain-conflicting sets of disks need to be executed sequentially, while operations on domain-disjoint sets can be executed concurrently.

Imagine a straw-man algorithm in which the set of disks is selected with a simple greedy algorithm: all 1,152 disks are put into a list, one is randomly selected and all drives that have a domain-conflict with it are removed from the list, and this repeats until 18 disks have been chosen. This is very simple, but yields very poor concurrency. If you take two groups, a and b, of size g populated using this algorithm then each disk in b has a probability proportional to g of conflict with group a. Therefore, the probability for a and b to be domain-conflicting is proportional to g^2.

The challenge with more complex data layout is to minimize the probability of domain conflicts while taming the computational complexity of determining the set of disks to be used to store a blob. The number of combinations of 18 out of 1,152 disks, C_{18}^{1152}, is large.

In order to handle the complexity we divided the disks into l groups, such that each disk is a member of a single group and all disks within a group are domain-disjoint, so they can be spinning concurrently. The group abstraction then removes the need to consider individual disks

[1] Non-systematic codes could also be used if beneficial [3].

or constraints. The complexity is now determining if l^2 pairs of logical groups are domain-disjointed or not, rather than C_{18}^{1152} sets of disks.

To improve concurrency compared to the straw-man algorithm, we enforce that if one disk in group *a* collides with group *b*, *all* the disks in *a* collide with *b*. In other words we make groups either fully colliding or fully disjoint, which reduces the collision probability from being proportional to g^2 to being proportional to g. This is close to the lower bound on the domain-collision probability for the Pelican hardware because g is the number of cooling domains used by a group (only one disk can be active per cooling domain and disks within a group are domain-disjoint).

Figure 2 shows a simplified example of how we assign disks to groups. The black squares show one group of 12 disks, the red-and-white squares another group. Notice they collide in all their power and cooling domains and so both groups cannot be spinning simultaneously. We start with the black group and generate all its rotations, which defines 12 mutually-colliding groups: the light-blue squares. These groups all fully collide, and we call this set of groups a *class*. Within a class only one group can be spinning at a time because of the domain conflicts. However the remaining domains are not conflicted and so available to form other classes of groups which will not collide with any of the 12 groups in the first class. By forming classes of maximally-colliding groups we reduce collisions between the other remaining groups and so greatly improve the available concurrency in the system.

Selecting l is reasonably straightforward: we wish to maximize (to increase scheduling flexibility) the number l of groups of size g given $l \times g = 1152$ and with $g >= k + r$ (groups have to be large enough to store a stripe). In the current implementation we use $g = 24$ rather than $g = 18$ so that a blob can always entirely reside within a single group even after some disks have failed. Stripe stacks stored on failed drives are initially regenerated and stored on other drives in the group. Hence $l = 48$; the 48 groups divide into 4 classes of 12 groups, and each class is independent from the others.

Using groups for the data layout has several benefits: *(i)* groups encapsulate all the constraints because disks in a group are domain-disjointed by definition; *(ii)* groups define the concurrency that can be achieved while servicing a queue of requests: groups from the same class have disks that share domain-conflicts and so need to be serviced sequentially, while groups from different classes have disks that are all domain-disjoint so can be serviced concurrently; *(iii)* groups span multiple failure domains: they contain disks distributed across the trays and all backplanes; and *(iv)* groups reduce time required to recover from a failed disk because all the required data is contained within the group.

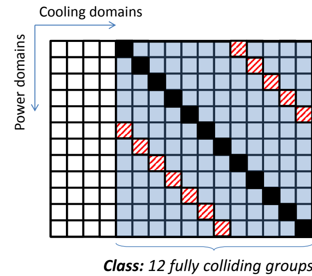


Figure 2: Two fully colliding groups.

A blob is written to $k + r$ disks in a single randomly selected group. The group has 24 disks but the blob only needs 18 disks. To select the disks to store the blob we split the group's disks into six sets each containing disks from the same backplane failure domain. The six sets are then ordered on spare capacity and the three disks with the highest spare capacity are selected. As we will show this simple approach achieves a high disk utilization.

Pelican preserves the group arrangement even when disks fail. A disk failure triggers rebuilds of all the blobs that stored a stripe stack on it. Rebuilding a single blob requires reading at least k stripe stacks and regenerating the missing stripe stack(s) across other disks in the group. By requiring blobs to be entirely within a group we ensure that rebuild operations use disks which are all spinning concurrently. The alternative would require buffering data in memory and spinning disks up and down during rebuild, which would slow down rebuild.

We use an off-rack metadata service called the *catalog* which is durable and highly available. Once disks have been chosen for a write request, the catalog is updated. It holds the mapping from a blob key to the 18 disks and group which store the blob, and other ancillary metadata. The catalog is modified during write, rebuild and delete requests, and information is looked up during read requests.

Using groups to abstract away the underlying hardware constraints is an important simplification for the IO scheduler: it needs simply consider which class the group is in rather than the constraints on all the drives. As we showed, increasing the number of groups which totally collide also increases the number of independent groups leading to better throughput and lower latency for operations. In the next section, we describe the IO scheduler in detail.

2.2.3 IO scheduler

In Pelican spin up is the new seek latency. Traditional disk schedulers have been optimized to re-order IOs in order to minimize *seek* latency overheads [14, 16]. In

Pelican we need to re-order requests in order to minimize the impact of *spin up* latency. The four classes defined by the data layout are domain-disjoint and are thus serviced independently. For each class, we run an independent instance of the scheduler that only services requests for its class. It has no visibility of requests on other classes and therefore the re-ordering happens at a class-level.

Traditional IO re-ordering attempts to order the requests to minimize the disk's physical head movement; every IO in the queue has a *different* relative cost to every other IO in the queue. We can define a cost function $c_d(h_l, IO_1, IO_2)$ which, given two IO requests IO_1 and IO_2 and the expected head position h_l, calculates the expected time cost of servicing IO_1 and then IO_2. This cost function will take into account the location of data being read or written, the rotational speed of the disk platters and the speed at which the head can move. This is a continuous function and $c_d(h_l, IO_1, IO_2) \neq c_d(h_l, IO_2, IO_1)$. In contrast, in Pelican there is a fixed constant cost of spinning up a group, which is independent of the current set of disks spinning. The cost function is $c_p(g_a, IO_g)$ where g_a is the currently spinning group and an IO has a group associated with it, so IO_g refers to an IO operation on group g. The cost function is binary, and if $g_a = g$ then the cost is zero, else it is 1.

We define the cost c of a proposed schedule of IO request as the sum of $c_p()$ for each request, assuming that the g of the previous request in the queue is g_a for the next request. Only one group out of the 12 can be spun up at any given time, and if there are q requests queued, and we use a FIFO queue, then $c \approx 0.92q$, as there is a probability of 0.92 that two consecutive operations will be on different groups. Note that there is an upper bound $c = q$, where every request causes a group to spin up.

The goal of the IO scheduler is to try to minimize c, given some constraint on the re-ordering queuing delay each request can tolerate. When $c \approx q$ then Pelican yields low throughput, as each spin up incurs a latency of at least 8 seconds. This means that in the *best case* only approximately 8 requests are serviced *per minute*. This also has the impact that, not only is throughput low, but the queuing latency for each operation will be high, as requests will be queued during the spin ups for the earlier requests in the queue.

Another challenge is ensuring that the window of vulnerability post-failures is controlled to ensure the probability of data loss is low. A disk failure in a group triggers a set of rebuild operations to regenerate the lost stripe-stacks. This rebuild requires activity on the group for a length of time equal to the data on the failed disk divided by the disk throughput (e.g., 100 MBps) since $k + r - 1$ stripe-stack reads and 1 stripe-stack write proceed concurrently. During the rebuild Pelican needs to service other requests for data from the affected group as well as other groups in the same class. Simply prioritizing rebuild traffic over other requests would provide the smallest window of vulnerability, but would also cause starvation for the other groups in the class.

The IO scheduler addresses these two challenges using two mechanisms: *request reordering* and *rate limiting*. Internally each scheduler instance uses two queues, one for rebuild operations the other for all other operations. We now describe these two mechanisms.

Reordering. In each queue, the scheduler can reorder operations independently. The goal of reordering is to batch sets of operations for the same group to amortize the group spin up latency over the set of operations. Making the batch sizes as large as possible minimizes c, but increases the queuing delay for some operations. To quantify the acceptable delay we use the delay compared to FIFO order.

Conceptually, the queue has a timestamp counter t that is incremented each time an operation is queued. When an operation r is to be inserted into the queue it is tagged with a timestamp $t_r = t$ and is assigned a reordering counter $o_r = t$. In general, $o_r - t_r$ represents the absolute change in ordering compared to a FIFO queue. There is an upper bound u on the tolerated re-ordering, and $o_a - t_a \leq u$ must hold for all operations a in the queue. The scheduler examines the queue and finds l, the last operation in the same group as r. If no such operation exists, r is appended to the tail of the queue and the process completes. Otherwise, the scheduler performs a check to quantify the impact if r were inserted after l in the queue. It considers all operations i following l. If $o_i + 1 - t_i \leq u$ no longer holds for any i, then r is appended to the tail of the queue. Otherwise all o_i counters are incremented by one, and r is inserted after l with $o_r = t_r - |i|$ where $|i|$ is the number of requests i, which r has overtaken.

To ease understanding of the algorithm, we have described the u in terms of the number of operations, which works if all the operations are for a uniform blob size. In order to support non-uniform blob sizes, we operate in wall clock time, and estimate dynamically the time each operation will be serviced, given the number of group spin ups and the volume of data to be read or written by operations before it in the queue. This allows us to specify u in terms of wall clock time.

This process is greedily repeated for each queued request, and guarantees that: *(i)* batching is maximized unless it violates fairness for some requests; and *(ii)* for each request the reordering bound is enforced. The algorithm expresses the tradeoff between throughput and fairness using u, which controls the reordering. For example, setting $u = 0$ results in a FIFO service order providing fairness, conversely setting $u = \infty$ minimizes the number of spin ups which increases throughput, but also means the request queuing delay is unbounded.

Rate limiting. The rate at which each queue is serviced is then controlled, to manage the interference between the rebuild and other operations. In particular, we want to make sure that the rebuild traffic gets sufficient resources to allow it to complete the rebuild within an upper time bound, to allow us to probabilistically ensure data durability if we know the AFR rates of the hardware.

The scheduler maintains two queues, one for the rebuild operations and one for other operations. We use a weighted fair queuing mechanism [6, 12] across the two queues which allows us to control the fraction of resources dedicated to servicing the rebuild traffic.

The approximate time to repair after a single disk failure is $x/t \times 1/w$ where x is the amount of data on the failed disk, t is the average throughput of a single disk (e.g., 100 MB/s) and w is the fraction of the resources the scheduler allocates to the rebuild.

3 Implementation

Early experience with the hardware has highlighted a number of other issues with right-provisioning. The first is to ensure that we do not violate the cooling or power constraints from when power is applied until the OS has finished booting and the Pelican service is managing the disks. We achieve this by ensuring the disks do not spin up when first powered, as done by RAID enclosures. Our first approach was to float pin 11 of the SATA power connector, which means that the drive spins up only when it successfully negotiates a PHY link with the HBA. We modified the Windows Server disk device driver to keep the PHY disabled until the Pelican service explicitly enables it. Once enabled the device driver announces it to the OS as normal, and Pelican can start to use it. However, this added considerable complexity, and so we moved to use Power Up In Standby (PUIS) where the drives establish a PHY link on power up, but require an explicit SATA command to spin up. This ensures disks do not spin without the Pelican storage stack managing the constraints.

At boot, once the Pelican storage stack controls the drives, it needs to mount and check every drive. Sequentially bring each disk up would adhere to the power and cooling constraints, but provides long boot times. In order to parallelize the boot process we exploit the group abstraction. We generate an initialization request for each group and schedule these as the first operation performed on each group. We know that the disks within the groups are domain-disjoint, so we can perform initialization concurrently on all 24 disks in the group. If there are no user requests present then four groups (96 disks) are concurrently initialized, initializing the entire rack takes the time required to sequentially initialize 12 disks (less than 3 minutes). External read and write requests can be concurrently scheduled, with the IO scheduler effectively on-demand initializing groups, amortizing the spin up time. The first time a disk is seen, Pelican writes a fresh GPT partition table and formats it with an NTFS filesystem.

During early development we observed unexpected spin ups of disks. When Pelican spins down a disk, it drains down IOs to the disk, unmounts the filesystem, and then sets the OFFLINE disk attribute. It then issues a STANDBY IMMEDIATE command to the disk, causing it to spin down. Disks would be spun up without a request from Pelican, due to services like the SMART disk failure predictor polling the disk. We therefore added a special *No Access* flag to the Windows Server driver stack that causes all IOs issued to a disk marked to return with a "media not ready" error code.

We also experienced problems with having many HBAs attached to the PCIe bus. The BIOS is responsible for initial PCI resource allocations of bus numbers, memory windows, and IO port ranges. Though neither the HBAs nor the OS require any IO ports, a small number are exposed by the HBA for legacy purposes. PCI requires that bridges decode IO ports at a granularity of 4 kB and the total IO port space is only 64 kB. We saw problems with BIOS code hanging once the total requirements exceeded 64 kB instead of leaving the PCI decode registers disabled and continuing. We have a modified BIOS on the server we use to ensure it can handle all 72 HBAs.

4 Evaluation

This section evaluates Pelican and in particular quantifies the impact of the resource right-provisioning on performance. We have a prototype Pelican rack with 6 chassis and a total of 1,152 disks. Our prototype uses archival class disks from a major disk manufacturer. Six PCIe uplinks from the chassis backplanes are connected to a single server using a Pelican PCIe aggregator card. The server is an HP ProLiant DL360p Gen8 with two eight-core Intel Xeon E5-2665 2.4GHz processors, 16 GB DRAM, and runs Windows Server 2012 R2. The server has a single 10 Gbps NIC. In order to allow us to evaluate the final Pelican configuration with two servers and to compare to alternative design points, we have developed a discrete event-based Pelican simulator. The simulator runs the same algorithms and has been cross-validated against the storage stack running on the full rack.

4.1 Pelican Simulator

The discrete event simulator models the disks, network and PCIe/SATA physical topology. In order to ensure

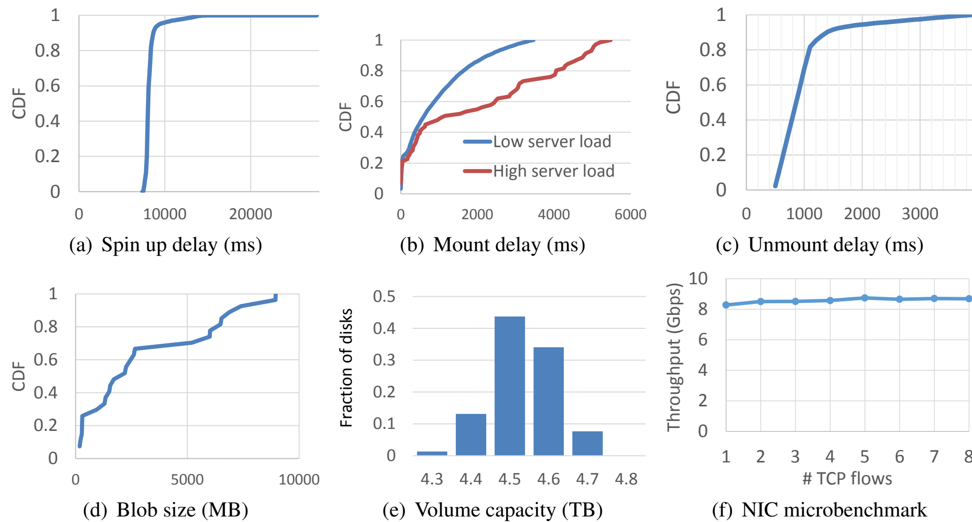


Figure 3: Distributions and parameters used in the simulation.

that the simulator is accurate we have parameterized the simulator using micro-benchmarks from the rack. We have then also cross-validated the simulator against the rack for a range of workloads.

We measure the spin up delays (shown in Figure 3(a)) and delays for mounting and unmounting a drive once it is spun up (shown in Figure 3(b) and 3(c)). For the spin up and unmounts delays we spun up and down disks 100,000 times, and measured the spin up and unmount latency. We observe that the volume mount delays are dependent on the load of the system. Under heavy load, the probability that at least one disk in a group is a straggler when all disks in the group are being mounted is much higher than when under low load. We therefore generate the mount delay distributions by taking samples of mount latencies during different load regimes on the Pelican. Figure 3(b) compares the distributions taken during high and low loads. In simulation, the mount delays are sampled from the distribution that corresponds to the current load.

For the disk throughput we measured the actual throughput of the disks in a quiescent system, and configure the simulator with an average disk throughput of 105 MB/s. Seeks are simulated by including a constant latency of 4.2ms for all disk accesses as specified in the disk data-sheet. In Pelican, seek latency has negligible impact on performance. Reads are for large blobs, and even though they are striped over multiple disks, each stripe stack will be a single contiguous file on disk. Further, as we can see from Figure 3(a) and 3(b) the latency of spinning the disk up and mounting the volume heavily dominate over seek time. The simulator uses a distribution of disk sizes shown in Figure 3(e). The capacities shown are for the drive capacity after formatting with NTFS. Finally, the Pelican is not CPU bound so we do not model CPU overheads.

In order to allow us to understand the performance effects of right-provisioning in Pelican, we also simulate a system organized like Pelican but with full provisioning for power and cooling which we denote as *FP*. In the FP configuration disks are *never spun down*, but the same physical internal topology is used. The disks are, as with Pelican, partitioned into 48 groups of 24 disks, however, in the FP configuration all 48 groups can be concurrently accessed. There is no spin up overhead to minimize, so FP maintains a queue per group and services each queue independently in FIFO order. FP represents an idealized configuration with no resource constraints and is hard to realize in practice. In particular, due to the high disk density in Pelican, little physical space is left for cooling and so forth.

In both configurations we assume, since the stripe stacks are contiguous on a disk, that they can be read at full throughput once the disk is spun up and mounted. The simulator models the PCIe/SATA and network bandwidth at flow level. For congested links the throughput of each flow is determined by using max-min fair sharing. As we will show in Section 4.4 we are able to cross-validate the Pelican simulator with the Pelican rack with a high degree of accuracy.

4.2 Configuration parameters

In all experiments the Pelican rack and simulator are configured with 48 groups of 24 disks as described. The groups are divided into 4 classes, and a scheduler is used per class. Blobs are stored using a 15+3 erasure encoding so each blob has 15 data blocks and 3 redundancy

blocks, all stored on separate disks. The maximum queue depth per scheduler is set to 1000 requests, and the maximum reordering allowance is set to 500 GB. For the cross-validation configuration, all 4 schedulers are run by a single server with one 10 Gbps NIC. In the rest of the evaluation, the rack has two servers such that each server runs two schedulers. Each server is configured with 20 Gbps network bandwidth, providing an aggregate network throughput of 40 Gbps for the entire Pelican rack.

4.3 Workload

We expect Pelican to be able to support a number of workloads, including archival workloads that would traditionally use tape. Pelican represents a new design point for cloud storage, with a small but non-negligible read latency and a limited aggregate throughput. We expect tiering algorithms and strategies will be developed that can identify cold data stored in the cloud that could be serviced by a Pelican. Due to the wide range of workloads we would like to support, we do not focus on a single workload but instead do a full parameter sweep over a range of possible workload characteristics. We generate a sequence of client read requests using a Poisson process with an average arrival rate $1/\lambda$, and vary $\lambda = 0.125$ to 16. For clarity, in the results we show the average request rate per second rather than the λ value. As write requests are offloaded to other storage tiers, we assume that servicing read requests is the key performance requirement for Pelican and, hence focus on read workloads. The read requests are randomly distributed across all the blobs stored in the rack. Unless otherwise stated requests operate on a blob size of 1 GB, to allow the metric of requests per second to be easily translated into an offered load. Some experiments use a distribution of blob sizes which is shown in Figure 3(d); a distribution of VHD image sizes from an enterprise with mean blob size of 3.3 GB.

In all simulator experiments, we wait for the system to reach steady state, then gather results over 24 simulated hours of execution.

4.4 Metrics

Our evaluation uses the following metrics:

Completion time. This is the time between a request being issued by a client and the last data byte being sent to the client. This captures the queuing delay, spin up latency and the time to read and transfer the data.

Time to first byte. The time between a request being issued by a client and the first data byte being sent to the client. This includes the queuing delay and any disk spin up and volume mount delays.

Service time. This is the time from when a request is dequeued by the scheduler to when the last byte associated with the request is transferred. This includes delays due to spinning up disks and the time taken to transfer the data, but excludes the queuing delay.

Average reject rate. These experiments use an open loop workload; when the offered load is higher than the Pelican or FP system can service, requests will be rejected once the schedulers' queues are full. This metric measures the average fraction of requests rejected. The NIC is the bottleneck at 5 requests per second for 40 Gbps. Therefore in all experiments, unless otherwise stated, we run the experiment to a rate of 8 request per second.

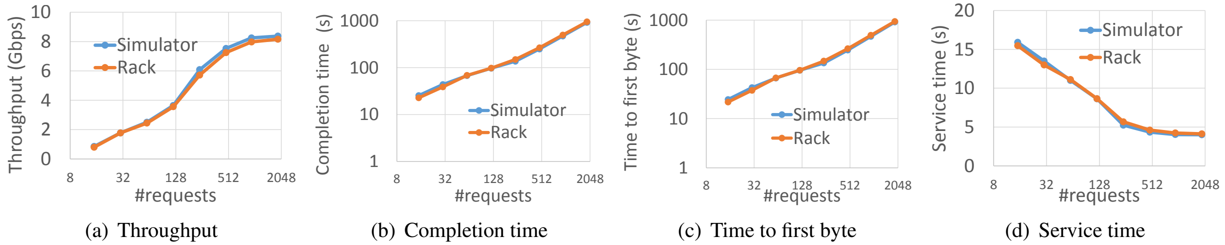
Throughput. This is the average rack network throughput which is calculated as the total number of bytes transferred during the experiment across the network link divided by the experiment duration.

First we cross-validated the simulator against the current Pelican rack. The current prototype rack can support only one disk spinning and one disk spinning up per tray, rather than two disks spinning up. The simulator is configured with this restriction for the cross-validation. The next revision of the rack and drives will be able to support two disks spinning up per tray. We configure the simulator to match the prototype Pelican rack hardware. A large number of experiments were run to test the algorithmic correctness of the simulator against the hardware implementation.

During the initial experiments on the real platform, we noticed that the NIC was unable to saturate the 10 Gbps NIC. Testing the performance of the NIC in isolation on an unloaded server using TTCP [10] identified that it had a peak throughput of only 8.5 Gbps. Figure 3(f) shows the network throughput as a function of the number of TCP flows. Despite our efforts to tune the NIC, the throughput of the NIC never exceeded 8.5 Gbps. For the cross-validation we configure the simulator to use an 8.5Gbps NIC.

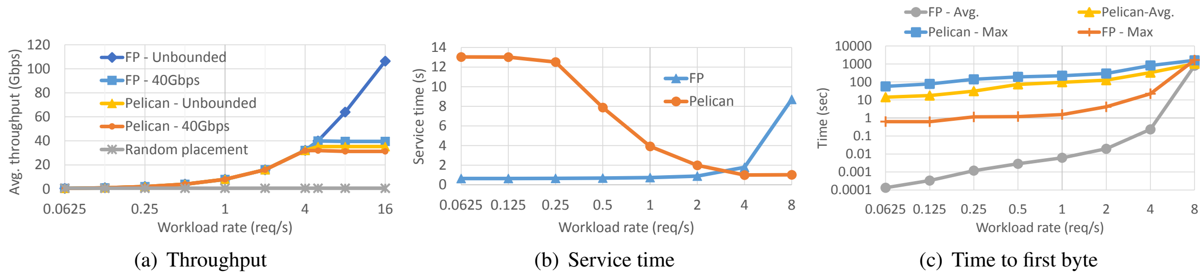
We ran a set of performance evaluation experiments where we generated a set of trace files consisting of a burst of b read requests for 1GB blobs uniformly distributed over 60 seconds, where we varied b from 15 to 1,920. We created a test harness that runs on a test server and reads a trace file and performs the read operations using the Pelican API running on the Pelican rack. Each experiment ran until all requests had been serviced. We replayed the same trace in the simulator. In both cases we pre-loaded the Pelican with the same set of blobs which are read during the trace.

To cross-validate we compare the mean throughput, request completion times, service times and times to first byte for different numbers of requests. The comparison is summarized in Figure 4. Figures 4(a) to 4(d) re-



(a) Throughput (b) Completion time (c) Time to first byte (d) Service time

Figure 4: Cross-validation of simulator and Pelican rack.



(a) Throughput (b) Service time (c) Time to first byte

Figure 5: Base performance of Pelican.

spectively show average throughput of the system, and average per-request completion time, time to first byte and service time as a function of the number of requests. These results show that the simulator accurately captures the performance of the real hardware for all the considered metrics. Traditionally, simulating hard disk-based storage accurately is very hard due to the complexity of the mechanical hard drives and their complex control software that runs on the drive [17]. In Pelican, the overheads are dominated by the spin up and disk mount latencies. The IO pattern at each disk is largely sequential, hence we do not need to accurately simulate the low-level performance of each physical disk.

All further results presented for Pelican and FP use this cross-validated simulator.

4.5 Base performance

The first set of experiments measure the base performance of Pelican and FP. Figure 5(a) shows the throughput versus the request rate for Pelican, FP and for the straw-man random layout (described in Section 2.2.2).

The straw-man performs poorly because the random placement means that the probability of being able to concurrently spin up two sets of disks to concurrently service two requests is very low. Across all request rates, it never achieves a throughput of more than 0.7 Gbps. The majority of requests are processed sequentially with group spin ups before each request. The latency of spinning up disks dominates and impacts throughput.

In Figure 5(a) we can also see that the throughput for

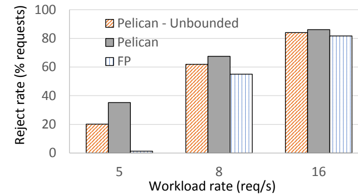


Figure 6: Reject rates versus workload request rate.

the default configurations of Pelican and FP configured with two servers each with 20 Gbps network bandwidth (labelled 40 Gbps in Figure 5(a)) is almost identical up to 4 requests per second. These results show that, for throughput, the impact of spinning up disks is small: across all the 40 Gbps configurations Pelican achieves a throughput within 20% of FP. The throughput plateaus for both systems after 5 requests per second at 40 Gbps which is the aggregate network bandwidth.

Figure 5(a) also shows the results for Pelican and FP configured with unbounded network bandwidth per server. This means the network bandwidth is never the bottleneck resource. This clearly shows the impact of right-provisioning. FP is able to utilize the extra resources in rack when the network bandwidth is no longer the bottleneck, and is able to sustain a throughput of 106 Gbps. In contrast, Pelican is unable to generate load for the extra network bandwidth because it is right-provisioned and configured for a target throughput of 40 Gbps.

Figure 6 shows the average reject rates for all the configurations under different loads, except for FP with un-

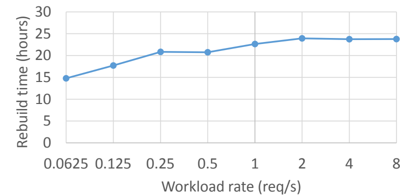
bounded bandwidth as it never rejects a request. No configuration rejects a request for rates lower than 5 requests per second. At 5 requests per second, Pelican and Pelican unbounded start to become overloaded and reject 35% and 21% of the requests respectively. FP also marginally rejects about 1% of requests because request queues are nearly full and minor fluctuations in the Poisson arrival rate occasionally cause some requests to be rejected. As the request rates increase the rejection rate increase significantly, with more than half of the submitted requests rejected.

The results above show that Pelican can achieve a throughput comparable to FP. The next results explore the per request service time. Figure 5(b) shows the average request service time as a function of the request rate. For FP when the offered load is low, there is no contention on the network, and hence the service time is simply the time taken to read the blob from the disks. Hence, for 1 GB blobs the lower bound on service time is approximately 0.65 seconds, given the disk throughput of 105 MB/s. As the offered load increases, concurrent requests get bottlenecked on the network, leading to the per-request service time increasing.

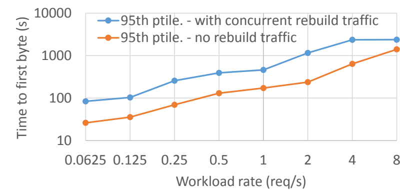
For Pelican, request rates in the range of 0.0625 and 0.25 per second, results in most requests incurring a spin up delay because requests are uniformly distributed across groups, and there are only 4 groups spun up at any time out of 48, so the probability of a request being for a group already spinning is low. As the request rate increases, the probability of multiple requests being queued for same group increases, and the average service time decreases. Above 0.25 requests per second there is sufficient number of requests being queued for the scheduler to re-order them to reduce the number of group spin ups. Although requests will not be serviced in the order they arrive, the average service time decreases as multiple requests are serviced per group spin up. Interestingly, for a rate of 4 requests per second and higher the service time for Pelican drops below FP, because in FP 48 requests are serviced concurrently versus only 4 in Pelican. The Pelican requests therefore obtain a higher fraction of the network bandwidth per request and so complete in less time.

Next we explore the impact on data access latency of needing to spin up disks. Figure 5(c) shows the average and maximum time on a log scale to first byte as a function of request rate for Pelican and FP. Under low offered load the latency is dominated by the disk seek for FP and by spin up for Pelican, so FP is able to get the first byte back in tens of milliseconds on average, while Pelican has an average time to first byte of 14.2 seconds even when idle. The maximum times show that FP has a higher variance, due to a request that experiences even short queuing delay being impacted by the service time of requests scheduled before it. However, the maximum for FP is over an order of magnitude lower than the mean for Pelican, until the request rate is 5 requests per second and are bottlenecked on the network bandwidth.

Overall, the results in Figure 5 show that Pelican can provide a high utilization with reasonable latency. In an unloaded Pelican blobs can be served quickly, whereas under heavy load, high throughput is delivered.



(a) Rebuild time



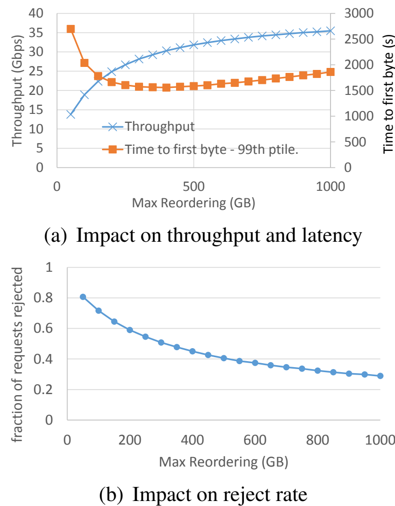
(b) Impact on client requests

Figure 7: Scheduler performance.

4.5.1 Impact of disk failure

Next, we evaluate how recovering from a disk failure impacts the time to first byte for concurrent client requests. We also evaluate the time Pelican takes to rebuild all lost blobs. The experiment is the same as that of the previous section, except that we mark one disk as failed once the system reaches steady state. The disk contains 4.28 TB of data and 64,329 blobs stored in the group have a stripe stack on the failed disk. For each blob $\geq k$ undamaged stripe stacks are read and the regenerated stripe stack is written to the same group, which has sufficient spare capacity distributed across the other disks in the group. The scheduler rate-limits client requests to ensure that the rebuild has at least 50% of the throughput.

Figure 7(a) shows the time to rebuild and persist all the blobs versus the client request rate. At low client request rates the offered client load is low enough that it does not use all the resources available to it, and due to work conservation, the time to recover is low. As the offered client traffic increases, the time taken to recover from the disk failure grows. The rebuild time plateaus below 24 hours, when both the rebuild and client request use their allocated 50%. Figure 7(b) shows the time to first byte for the 95th percentile of client requests versus client request rate with and without rebuild requests. The rate limiting increases the 95th percentile time to first byte for



(a) Impact on throughput and latency

(b) Impact on reject rate

Figure 8: Cost of fairness.

client requests by a factor of 2.2 at high workload rates. The rebuild traffic only affects client requests in the class impacted by the disk failure. Other classes observe no performance impact.

4.6 Cost of fairness

The next experiments evaluate the impact on fairness by the Pelican scheduler. We fix the client workload at 5 read requests per second, the maximum rate which the network could sustain. We ran the experiment varying u, the tolerance on re-ordering, between 50 and 1000 GB. Figure 8(a) shows the throughput and 99th percentile of time to first byte as a function of u. We show the 99th percentile for time to first byte to quantify the impact on the tail, those requests most impacted by queuing delay.

Changing u from 50 to 350 GB nearly doubles the throughput of the system. Increasing u beyond this has only a modest impact on throughput. This is because the fraction of time spent spinning up drives is low therefore the majority of time is being spent reading data. When u is below 350 GB, the 99th percentile of time to first byte is dominated by the queuing delay. A significant fraction of the time is spent spinning disks up, impacting both throughput and latency of request. Hence increasing reordering improves throughput and reduces the queuing delay. At above 350GB the 99th percentile for time to first byte increases, despite the slight increase in throughput. This is slightly counter intuitive, until you remember that as u increases we are allowing requests to be queued for longer. This is then reflected when looking at the 99th percentile. This also impacts the number of requests that are rejected. Figure 8(b) shows the reject rate of the system as a function of u. Changing u from 50 to 1000 GB reduces the percentage of rejected requests from nearly 85% to approximately 25% due to increased throughput.

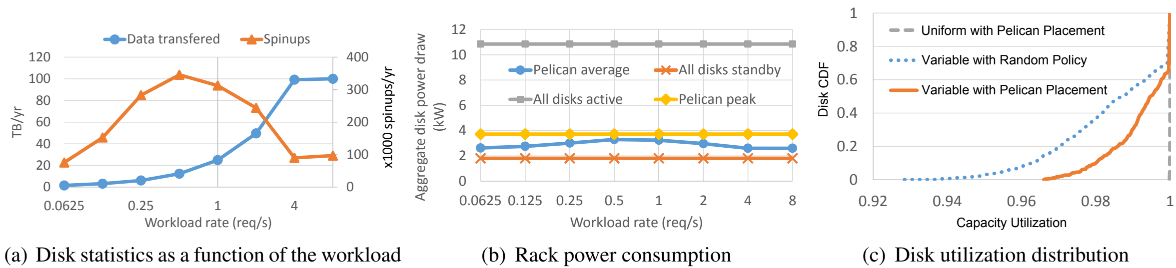
4.7 Disk Lifetime

An obvious concern is the impact on the lifetime of the disks of spinning them up and down. Also, the new class of archival drive that systems like Pelican use are rated for a number of terabytes read and written per year. We therefore ran an experiment to determine the average number of spin ups per year as well as the expected number of terabytes transferred. Figure 9(a) shows the average number of spin ups and terabytes transferred per year as a function of the workload rate. The average data transferred increases with the request rate and peaks at 99 TB per year when the request rate saturates the system. This is within the specification for the new generation of archival drives. Currently in the prototype Pelican we do not do any proactive background data scrubbing, which is common in storage systems to check for integrity issues. Background scrubbing increases the volume of data transferred per disk, which in itself impacts the disk AFR. We are currently long-term empirically testing the drives and plan to add scrubbing functionality once we have a better understanding of the drives longer term performance.

The number of spin ups is also shown in Figure 9(a); interestingly the peak is at 0.5 requests per second. Below this rate the number of spin ups grows with load as the scheduler has little opportunity to reorder requests. Above this rate the number of spin ups decreases because the queues are long enough for the scheduler to perform reordering. At 4 requests per second the scheduler hits the maximum reordering limit and the number of spin ups remains constant as the request rate increases further. We believe that the archival class of drive that we are using can tolerate this many spin up cycles per year with minimal impact on the AFR. It should be noted that these are controlled head park and unpark operations triggered by issuing a SATA command to the drive, which is then allowed to park the head. They are not induced by sudden power failure. When a disk is spun down, all the electronics in the drive are still powered and operating.

4.8 Power Consumption

We now quantify the power savings resulting from power right-provisioning. The prototype Pelican hardware enables us to measure the power draw of each tray independently. We ran an experiment in which we sequentially spun up then down every disk in a tray, sampling the tray power draw. This allows us to estimate the average power draw of a disk in standby, spinning up and active states. The tray power is dominated by the disks, so



(a) Disk statistics as a function of the workload (b) Rack power consumption (c) Disk utilization distribution

Figure 9: Power draw and disk usage.

we place all 16 disks in standby and estimate the power draw per disk as $\frac{1}{16}$th of the power draw of a tray. The average power draw for a disk active or spinning up is then computed using the average power draw of a tray with one disk in that test state minus the power draw of the other disks in standby. The power draws for the three disks states is: $P_{standby} = 1.56$ W, $P_{spinup} = 21.53$ W and $P_{active} = 9.42$ W.

We parameterize the simulator with these values and measure the power consumed by just the disks. Figure 9(b) shows the power draw of the 1,152 disks under different configurations as a function of the workload rate. The figure shows the average and peak power draw for Pelican. It also shows the lower bound when all drives in spun down in standby and, in order to allow comparison with a fully provisioned system, it shows the power draw for all drives spun up and active.

The average power draw of Pelican varies with the workload rate. The highest average power consumption which is 3.3 kW is reached when the number of spin ups is the highest, at around 0.5 requests per second because spin ups have the highest power draw. At this rate, there are sufficient requests to require frequent group spin ups, but not enough for extensive batching. For both lower and higher request rates, group spin ups are less frequent and the power consumption is close to 2.6 kW. Across all request rates the average Pelican power draw is between 3.6 and 4.1 times lower compared to the fully provisioned power draw with all disk spinning. Pelican peak power consumption is 3.7 kW and is reached when 96 drives are concurrently spinning up. For comparison, peak power draw is 3 times lower than all disks active. In the fully provisioned rack the peak power draw would be achieved if all 1,152 drives were concurrently spun up and would peak at 25.8kW. Of course, in practice some form of deferred spin up technique would be used.

4.9 Capacity Utilization

The final set of experiments evaluate the Pelican data layout algorithm with respect to capacity utilization, particularly with non-uniform disk capacities. We ran an experiment with a 100% write workload that stopped when the first write request was rejected because no group had sufficient remaining capacity to store the blob. We measure the per-disk utilization across the rack. For this experiment, the blobs sizes are selected using a distribution of VHD sizes from an internal company VHD store, with sizes from 200 MB to 9 GB with an average of 3.3 GB.

Figure 9(c) shows the CDF of per-disk utilization for three configurations. The solid line uses the Pelican placement policy with variable disk capacities, the dotted line is if the disks within a group are selected randomly from the subset that have sufficient spare capacity. Finally, the dashed line is for disks of equal size (set to the mean capacity of the disks used for variable). Comparing the Pelican approach to the random policy with variable capacity disks shows the benefit of Pelican approach.

To understand this more consider the rack utilization, rather than per-disk utilization. The total capacity utilization for an entire rack is 99.386% for Pelican with variable disk capacities as shown in figure 3(e) versus 99.998% for uniform disk capacities. At 1,152 disks, this implies that the equivalent of 7.07 disks are completely empty when using variable capacities when the first request is rejected. However, a traditional RAID system would have clipped all these drives to the minimum 4.2 TB. Therefore, compared to a 100% utilization at 4.2 TB, adapting to variable capacity is giving us $99.4 \times 4.55/4.2 = 107.7\%$ utilization.

5 Related Work

There has been extensive work on enabling disks to spin down under low load, including [5, 9, 2, 15]. Several of these systems proposed to modify the individual disk IO handling to increase periods of disk inactivity, allowing disks to spin down to save power [9, 13]. Write offloading [9] proposed allowing active disks to be used to buffer writes that were targeted against disks that were spun down. Only read requests for data not available required disks to be spun up, increasing the fraction of time disks could be spun down. Write-offloading makes no assumptions on the data layout used and worked at

the block level. Pergamum [13] is a MAID-like system; it incorporates NVRAM to handle meta-data, absorbing meta-data reads and writes as well as buffering writes to spun down disks. It also uses spin up tokens to limit peak power draw. These mechanisms could be exploited to support right-provisioning of power. However, unlike Pelican, Pergamum allows clients to select the disks being used to store their data. In Pelican the data layout is a function of the physical location of the disks which determines their power and cooling domains, and Pelican schedules these accesses appropriately.

Massive Arrays Of Idle Disks (MAID) [5] systems assume that the power and cooling resources are provisioned to support a peak performance. In contrast, Pelican is right-provisioned for the workload rather than for peak hardware performance.

Systems such as Rabbit [2] and Sierra [15] are power-proportional. In a power-proportional system the power consumption is proportional to the load. This is usually achieved through careful data layout schemes [15]. However, again most of these systems assume at peak performance all disks can be spun up. In many ways Pelican is also power-proportional, the number of disks spinning is a function of the workload, except the number of drives spinning never exceeds 8% of the disks.

Pergamum [13] also assumed a model where each disk was effectively connected directly to the network using an Ethernet port, with a small processor board mounted in front of each disk for local processing. This general model has been adopted by the Seagate Kinetic drives [11]. Pelican uses standard SATA archival drives and PCIe at the rack-scale, primarily to minimize costs. Further, this allows a single server to accurately control the drives and their states during boot time and during operation.

Finally, Amazon Glacier [1] and the Facebook cold data storage SKU [8] have never had public details released on their software stack to date. Facebook has released a design of the hardware through the Open Compute Project (OCP). Compared to the public information, Pelican has a higher disk density, lower system power consumption per disk, lower hardware cost, and provides smaller failure domains when compared to the OCP Cold Storage reference design.

6 Future Work

Pelican raises a number of interesting questions which we leave open for future work. The current Pelican software stack was co-designed with the hardware. This has the benefit that we are able to right-provision Pelican, but it has the drawback that the disk group assignment is very brittle with respect to hardware changes. For example, changing the cooling or power domains, or adding a new constraint, would require a redesign of the data layout and re-working of the IO scheduler. Further, designing the storage stack to work within the constraints is not trivial and took many months, and getting it wrong is not always immediately or trivially visible, for example violating the vibration domain. We do not know if we have an optimal design, simply we have one that seems to perform well. To address these issues we are currently developing tools to automatically synthesize the data layout and IO scheduling policies. We believe that we can encapsulate the underlying principles that we learnt when building this storage stack in a tool. This will enable rapid (automatic) optimisation of cold storage.

The other issue is that for many years the enterprise storage community has avoided spinning disks up and down, as it tends to yield higher failure rates, and even worse, correlated failures. We hope to gain understanding of this empirically over time. In particular, the spinning up and down of groups together, while helping improve performance when batching multiple requests, means all drives in the same group have an identical history. If we observe correlated failures, we can be more conservative and spin up only the 18 disks which are required to service a particular operation. We can also make sure that all groups spin up with a particular minimum frequency, as well as watch particular performance metrics for early signs of high wear. Many of these things we will only discover over time and we look forward to reporting them to the community.

7 Conclusion

Pelican is designed to support workloads where data stored is rarely read, often referred to as cold data. Pelican is unique in that the hardware has been designed to be right-provisioned. In this paper we have described and evaluated how these hardware limitations impact the data layout and IO scheduling. We have shown that the Pelican mechanisms are effective and compared against a fully provisioned system.

Acknowledgements

We would like to thank the following for their help: Cheng Huang, Brad Calder, Thierry Fevrier, Karan Mehra, Erik Hortsch and David Goebel. We'd like to thank our shepherd, Emin Gün Sirer, for his comments and support. We'd also like to thank the anonymous reviews for their insightful and helpful feedback.

References

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

[2] 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* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 217–228.

[3] ANDRÉ, F., KERMARREC, A.-M., MERRER, E. L., SCOUARNEC, N. L., STRAUB, G., AND VAN KEMPEN, A. Archiving cold data in warehouses with clustered network coding. In *Proceedings of EuroSys* (New York, NY, USA, Apr 2014), ACM.

[4] BLÖMER, J., KALFANE, M., KARP, R., KARPINSKI, M., LUBY, M., AND ZUCKERMAN, D. An XOR-Based Erasure-Resilient Coding Scheme. Tech. Rep. ICSI TR-95-048, University of California, Berkeley, August 1995.

[5] COLARELLI, D., AND GRUMWALD, D. Massive Arrays of Idle Disks for Storage Archives. In *IEEE Supercomputing* (July 2002).

[6] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queueing algorithm. In *Symposium Proceedings on Communications Architectures & Protocols* (New York, NY, USA, 1989), SIGCOMM '89, ACM, pp. 1–12.

[7] 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.

[8] MORGAN, T. P. Facebook loads up innovative cold storage datacenter. `http://www.enterprisetech.com/2013/10/25/facebook-loads-innovative-cold-storage-datacenter/`, October 2013.

[9] 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.

[10] Ntttcp. `http://gallery.technet.microsoft.com/NTttcp-Version-528-Now-f8b12769`, July 2013.

[11] SEAGATE. The seagate kinetic open storage vision. `http://tinyurl.com/noj7glm`, August 2014.

[12] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queueing using deficit round robin. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 1995), SIGCOMM '95, ACM, pp. 231–242.

[13] 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), USENIX Association, p. 1.

[14] TEOREY, T. J., AND PINKERTON, T. B. A comparative analysis of disk scheduling policies. In *Proceedings of the Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1971), SOSP '71, ACM, pp. 114–.

[15] THERESKA, E., DONNELLY, A., AND NARAYANAN, D. Sierra: Practical power-proportionality for data center storage. In *Proceedings of the Sixth Conference on Computer Systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 169–182.

[16] WORTHINGTON, B. L., GANGER, G. R., AND PATT, Y. N. Scheduling algorithms for modern disk drives. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 1994), SIGMETRICS '94, ACM, pp. 241–251.

[17] WORTHINGTON, B. L., GANGER, G. R., PATT, Y. N., AND WILKES, J. On-line extraction of scsi disk drive parameters. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1995), pp. 146–156.