Gecko: Contention-Oblivious Disk Arrays for Cloud Storage

Ji-Yong Shin* Mahesh Balakrishnan‡ Tudor Marian† Hakim Weatherspoon*
*Cornell University ‡Microsoft Research †Google

Abstract

Disk contention is increasingly a significant problem for cloud storage, as applications are forced to co-exist on machines and share physical disk resources. Disks are notoriously sensitive to contention; a single application's random I/O is sufficient to reduce the throughput of a disk array by an order of magnitude, disrupting every other application running on the same array. Log-structured storage designs can alleviate write-write contention between applications by sequentializing all writes, but have historically suffered from read-write contention triggered by garbage collection (GC) as well as application reads. Gecko is a novel log-structured design that eliminates read-write contention by chaining together a small number of drives into a single log, effectively separating the tail of the log (where writes are appended) from its body. As a result, writes proceed to the tail drive without contention from either GC reads or first-class reads, which are restricted to the body of the log with the help of a tail-specific caching policy. Gecko trades-off maximum contention-free sequential throughput from multiple drives in exchange for a stable and predictable maximum throughput from a single uncontended drive, and achieves better performance compared to native log-structured or RAID based systems for most cases. Our in-kernel implementation provides random write bandwidth to applications of 60 to 120MB/s, despite concurrent GC activity, application reads, and an adversarial workload.

1 Introduction

Modern data centers are heavily virtualized, with the compute and storage resources of each physical server multiplexed across a large number of applications. Two trends point to increased virtualization. The first is the emergence of cloud computing, where multiple tenants are routinely assigned to different cores on the same machine. The second trend is the increasing number of cores on individual machines, driven by the end of frequency scaling, which forces applications to co-exist on the same server. Virtualization enables applications to share machine resources without resorting to physical partitioning, allowing resources such as disk capacity or network bandwidth to be temporally multiplexed across many different tenants.

Unfortunately, virtualization leads to contention. In virtualized settings, applications are susceptible to the behavior of other applications executing on the same machine, network and storage infrastructure. In particular, contention in the storage subsystem of a single machine is a significant issue, especially when a disk array is shared by multiple applications running on different cores. In such a setting, an application designed for high I/O performance – for example, one that always writes or reads sequentially to disk – can perform poorly due to random I/O introduced by applications running on other cores [8]; later in this paper, we quantify this effect. In fact, even in the case where every application on the physical machine accesses storage strictly sequentially, the disk array can still see a non-sequential I/O workload due to the inter-mixing of multiple sequential streams [10]. Disk contention of this nature is endemic to any system design where a single disk array is shared by multiple applications running on different cores.

Existing solutions to mitigate the effects of disk contention revolve around careful scheduling decisions, either spatial or temporal. For instance, one solution to minimize interference involves careful placement of applications on machines [8, 9]. However, this requires the cloud provider to accurately predict the future I/O patterns of applications. Additionally, placement decisions are usually driven by a wide number of considerations, not just disk I/O patterns; these include data/network locality, bandwidth and CPU usage, migration costs, security concerns, etc. A different solution involves scheduling I/O to maintain the sequentiality of the workload seen by the disk array. Typically, this involves delaying the I/O of other applications while a particular application is accessing the disk array. However, I/O scheduling sacrifices access latency for better throughput, which may not be an acceptable trade-off for many applications.

A more promising approach is to build systems that are oblivious to contention by design. For instance, log-structured designs for storage – such as the log-structured filesystem (LFS) [18] – can support sequential or random write streams from multiple applications at the full sequential speed of the underlying media. Unfortunately, the Achilles' Heel of LFS is read-write contention caused by garbage collection (GC) [21, 13]; specifically, the random reads introduced by GC often interfere with first-

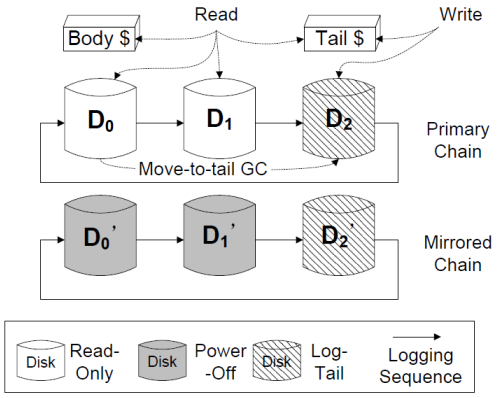

Figure 1: Chained Logging: all writes go to the tail drive of the chain, while reads are serviced mostly from the body of the chain or a cache. Mirrors in the body can be powered down.

class writes by the application, negating any improvement in write throughput. Additionally, LFS can also be subject to read-write contention from application reads; the original LFS work assumed that large caches would eliminate reads to the point where they did not interfere with write throughput. More recently, systems have emerged that utilize new flash technology to implement read caches or log-structured write caches [4] that can support contention-free I/O from multiple applications. However, this results in a highly stressful write workload for the flash drives that can wear them out within months [23].

In this paper, we propose Gecko, a new log-structured design for disk arrays. The key idea in Gecko is *chained logging*, in which the tail of the log – where writes occur – is separated from its body by placing it on a different drive. In other words, the log is formed by concatenating or chaining multiple drives. Figure 1 shows a chain of three drives, D_0, D_1 and D_2. On a brand new deployment, writes will first go to D_0; once D_0 fills up, the log spills over to D_1, and then in turn to D_2. In this state, new writes go to D_2, where the tail of the log is now located, while reads go to all drives. As space on D_0 and D_1 is freed due to overwrites on the logical address space, compaction and garbage collection is initiated. As a result, when D_2 finally fills up, the log can switch back to using free space on D_0 and D_1. Any number of drives can be chained in this fashion. Also, each link in the chain can be a mirrored pair of drives (e.g., D_0 and $D_0^{'}$) for fault-tolerance and better read performance.

The key insight in chained logging is that the sequential, contention-free write bandwidth of a single drive is preferable to the randomized, contention-affected bandwidth of a larger array. As with any logging design, chained logging ensures that write-write contention between applications does not result in degraded throughput, since all writes are logged sequentially at the tail drive of the chain. Crucially, chained logging also eliminates read-write contention between garbage collection (GC) activity and first-class writes by separating the tail of the log from its body. In the process, it trades off the maximum contention-free write throughput of the array – which is now limited to the sequential bandwidth of the tail drive of the chain – in exchange for stable, predictable write performance in the face of contention. In our evaluation, we show that a Gecko chain can operate at 60MB/s to 120MB/s under heavy write-write contention and concurrent GC activity, whereas a conventional log-structured RAID-0 configuration over the same drives collapses to around 10MB/s during GC.

To tackle read-write contention caused by application reads, Gecko uses flash and RAM-based caching policies that leverage the unique structure of the logging chain. All new writes to the tail drive in the chain are first cached in RAM, and then lazily moved to an SSD cache dedicated to the tail drive. As a result, reads on recently written data on the tail drive are served by the RAM cache, and reads on older data on the tail drive are served by the SSD tail cache. This caching design has two important properties. First, it is tail-specific: it prevents application reads from reaching the tail drive and randomizing its workload, thus allowing writes to proceed sequentially without interference from reads. Based on our analysis of server block-level traces, we found that a RAM cache of 2GB and an SSD cache of 32GB was sufficient to absorb over 86% of reads directed at the 512GB tail drive of a Gecko chain for all the workload combinations we tried. Second, it's two-tier structure allows overwrites to be coalesced in RAM before they reach the SSD cache; as we show in our evaluation, this can prolong the lifetime of the SSD by 2X to 8X compared to a conventional caching design.

Chained logging has other benefits. Eliminating read-write contention has the side-effect that writes no longer slow down reads. As a result, chained logs can exhibit higher read throughput for many workloads compared to conventional RAID variants, since reads are served by either the tail cache or the body of the log and consequently do not have to contend with write traffic. Chained logging can also be used to save power: when mirrored drives are chained together, half the disks in the body of the log can be safely switched off since they do not receive any writes. This lowers the read throughput of the log, but does not compromise fault-tolerance.

Importantly, Gecko is a log-structured block device rather than a filesystem; as a result, any filesystem or database can execute over it without modification. Historically, the difficulty of persistently maintaining metadata under the block layer has outweighed the benefits of block-level logging, forcing such designs to incur meta-

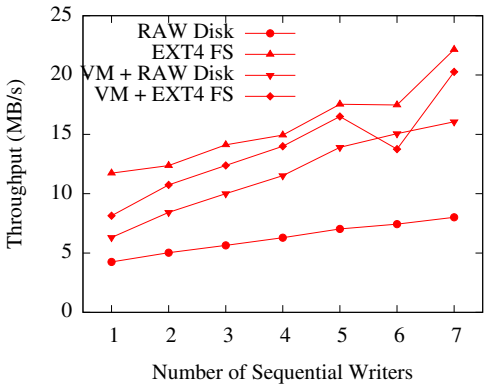

Figure 2: Throughput of 4-disk RAID-0 storage under N sequential writers + 1 random writer.

data seeks on disk or restricting them to expensive enterprise storage solutions that can afford battery-backed RAM or other forms of NVRAM [5, 6, 25, 19, 14]. Gecko is the first system to use a commodity MLC SSD to store metadata for a log-structured disk array; accordingly, it uses a new metadata scheme carefully designed to exploit the access characteristics of flash as well as conserve its lifetime.

This paper makes the following contributions. First, we propose the novel technique of chained logging, which provides the benefits of log-structured storage (obliviousness to write-write contention) without suffering from its drawbacks (susceptibility to read-write contention). Second, we describe the design of a block storage device called Gecko that implements chained logging, focusing on how the system utilizes inexpensive commodity flash for caching and persistence over the chained log structure. Third, we evaluate a software implementation of Gecko, showing that chained logging provides high, stable write throughput during GC activity, in contrast to log-structured RAID-0; it effectively prevents reads from impacting write throughput by using a tail-specific cache; and it outperforms log-structured RAID-0 in terms of both read and write performance on real workloads.

2 Motivation

In this section, we first motivate the problem of disk contention in virtualized data centers. We then provide the rationale for log-structured designs in such settings.

2.1 Disk Contention

Our focus is on settings where multiple applications share common disk infrastructure on a single physical machine. A common example of such a setting is a virtualized environment where multiple virtual machines (VMs) execute on a single machine and operate on filesystems that are stored on virtual disks. The guest OS within each VM is oblivious to the virtual nature of the underlying disk and the existence of other VMs on the same machine. In reality, virtual disks are implemented as logical volumes or files in a host filesystem. While performance isolation across VMs can be achieved by storing each virtual disk in a separate disk or disk array, this defeats the goal of virtualization to achieve efficient multiplexing of resources. Accordingly, it is usual for different virtual disks to reside on the same set of physical disks.

Disk virtualization leads to disk contention. A single badly behaved application that continually issues random I/O to a disk array can disrupt the throughput of every other application running over that array [8]. As machines come packed with increasing numbers of cores – and as cloud providers cram more tenants on a single physical box – it becomes increasingly likely that some application is issuing random I/O at any given time, disrupting the overall throughput of the entire system. In fact, throughput in such settings is likely to be sub-optimal even if every application on the system is well-behaved and perfectly sequential in its I/O behavior, since the physical disk array sees a mix of multiple sequential streams that is unlikely to stay sequential [10].

To illustrate these problems, we ran a simple experiment on an 8-core machine with 4 disks configured as a RAID-0 array. In the experiment, we ran multiple writers concurrently on different cores to observe the resulting impact on throughput. To make sure that the results were not specific to virtual machines, we ran the experiments with different levels of layering: processes writing to a raw volume (RAW Disk), processes writing to a filesystem (EXT4 FS), processes within different VMs writing to a raw volume (VM + RAW disk), and processes within different VMs writing to a filesystem (VM + EXT4 FS). In the absence of contention (i.e., with a single sequential writer), we were able to obtain 300 to 400MB/s of write throughput in this setup, depending on the degree of layering. Adding more sequential writers lowered throughput; with 8 writers, the system ran at between 120 and 300MB/s.

Figure 2 shows the impact on throughput of a single random writer when collocated with sequential writers. We show measurements of system throughput for increasing numbers of sequential writers, along with a single random writer issuing 4KB writes. For any number of sequential writers and any degree of layering, throughput is limited to less than 25MB/s, representing an order of magnitude drop compared to 300 to 400MB/s throughput without the random writer. One interesting point is that added layering improves throughput in the presence of a random writer; we believe this is due to scheduling intelligence in these layers that delays random I/O to improve sequentiality, a hypothesis borne out by observed I/O latencies.

This graph is not meant to be a comprehensive picture of contention in the cloud; rather, it illustrates how easy it is for a single application to disrupt system throughput in virtualized settings. Next, we look at how log-structured systems can help.

2.2 Flash and the Return of Log-Structured Systems

Log-structured filesystems were introduced in the 90s on the premise that the falling price of RAM would allow for large, inexpensive read caches. Accordingly, workloads were expected to be increasingly write-dominated, prompting designs such as LFS that converted slow random writes into fast sequential writes to disk.

Today, a similar argument can be made for flash instead of RAM. Flash drives have been steadily dropping in price; today, raw flash costs around $1 per GB, and SATA SSDs typically cost $2 per GB. Given this trend, it is tempting to imagine that flash will soon replace disk, or more pragmatically, act as a write cache for disk. Unfortunately, cheaper flash translates into less reliable flash, which in turn translates into limited device lifetime. The two ways of lowering flash cost – decreasing process sizes and cramming more bits per flash cell (i.e., MLC flash) – both result in much higher error rates, straining the ability of hardware ECC to provide disk-like reliability. As a result, lower costs have been accompanied by lower erase cycle thresholds, which determine the lifetime of the device when it is subjected to heavy write workloads. In other words, the cost per GB of flash has dropped, but not the cost per erase cycle; for example, today's MLC drives offer an order of magnitude fewer erase cycles compared to drives made from older SLC technology, while cutting price per GB by 25% to 50%.

On the other hand, read caches are a more promising use of flash. Unlike primary stores or write caches, read caches do not need to see every update immediately, but instead have leeway in deciding when (and whether) to cache data. For example, a read cache might wait for some time period before caching a newly written block in order for overwrites to be coalesced, extending flash longevity. It could also avoid caching data that's frequently overwritten but rarely read. Crucially, read caches do not need to be durable and hence the lower reliability of flash over time is not as much of a barrier to deployment; all that's required is a reliable mechanism to detect data corruption, which effectively translates into a cache miss.

Accordingly, our core assumption is nearly identical to that of the original LFS work: larger, effective (flash-based) read caches will result in write-dominated workloads. Unfortunately, simply using LFS under a flash-based read cache doesn't work, because of two key problems. First, as noted earlier, LFS is notorious for its garbage collection woes; GC reads (which are unlikely to be caught by a read cache) can contend with first-class writes, negating the positive effect of logging writes. Second, even a small fraction of random reads sneaking past the cache can interfere with write throughput. In other words, LFS effectively prevents write-write contention but is very susceptible to read-write contention, both from GC reads and first-class reads. Our goal in this paper is to build a log-structured storage design that prevents both write-write as well as read-write contention.

In addition to caching reads, MLC flash further acts as a catalyst for log-structured designs by providing an inexpensive, durable metadata store. A primary challenge for any log-structured system involves maintaining an index over the log. As a result, log-structured designs are usually found at layers of the stack that already require indices in some form, such as filesystems or databases. Designs at the block-level with a logging component (or indeed, any kind of indirection layer) have historically been hamstrung by seeks on on-disk metadata, or predicated on the availability of battery-backed RAM or NV-RAM [5, 6, 25]. Consequently, such designs have been restricted to expensive enterprise storage solutions. By providing an inexpensive means of durably storing an index and accessing it rapidly, MLC flash enables log-structured designs at lower layers of the stack, such as the block device.

3 Design

Gecko implements the abstraction of a block device, supporting reads and writes to a linear address space of fixed-size sectors. Underneath, this address space is implemented over a chained log structure, in which a single logical log is chained or concatenated across multiple drives such that the tail of the log and its body are on different drives. A new write to a sector in the address space is sent to the tail of the log; if it's an overwrite, the previous entry in the log for that sector is invalidated or trimmed. As the body of the log gets fragmented due to such overwrites on the address space, it is cleaned so that the freed space can be reused; importantly, this GC activity incurs reads on the body of the chained log, which do not interfere with first-class writes occurring at the tail drive of the log.

We first present the simplest possible instantiation of chained logging in Gecko, and then describe more sophisticated features. Gecko is implemented as a block device driver, occupying the same slot in the OS stack as software RAID; as with RAID, it can also be implemented in the form of a hardware controller. Gecko maintains an in-memory map (implemented as a simple array) from logical sectors on the supported address space to physical locations on the drives composing the array. In addition, it maintains an inverse map (also a

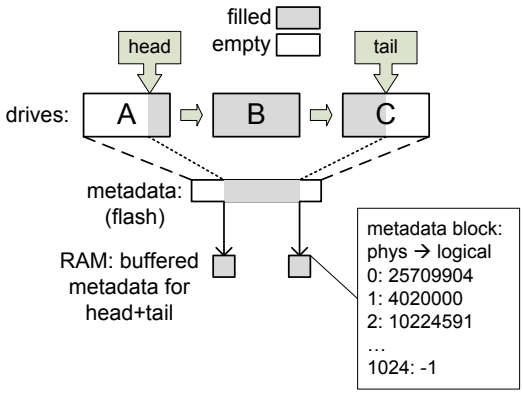

Figure 3: Metadata persistence in Gecko: mapping from physical to logical addresses is stored on flash, with actively modified head and tail metadata buffered in RAM.

simple array) to find the logical sector that a physical location stores; a special 'blank' value is used to indicate that the physical location does not contain valid data. Also, Gecko maintains two counters – one for the tail of the log and one for the head – each of which indexes into the total physical space available on the disk array.

When the application issues a read on a logical sector in the address space, the primary map is consulted to determine the corresponding physical location. When the application writes to a logical sector, the tail counter is checked and a write I/O is issued to the corresponding physical location on the tail drive. Both the primary map and the inverse map are then updated to reflect the linkage between the logical sector and the physical location, and the tail counter is incremented.

In the default form of GC supported by Gecko, data is constantly moved from the head of the chained log to its tail in order to reclaim space; we call this 'move-to-tail' GC. A cleaning process examines the next physical entry at the head of the log, checks if it is occupied by consulting the inverse map, and if so re-appends it to the tail of the log. It then increments the head and (if the entry was moved) the tail counter.

The basic system described thus far provides the main benefit of log chains – logging without interference from GC reads – but suffers from other problems. It does not offer tolerance to power failures or to disk failures. While GC writes do not drastically affect first-class writes, they do occur on the same drive as application writes and hence reduce write throughput to some extent. Further, the system is susceptible to contention between application reads and writes: reads to recently written data will go to the tail disk and disrupt first-class writes. Below, we discuss solutions to address these concerns.

3.1 Metadata

The total amount of metadata required by Gecko can easily fit into RAM on modern machines; to support a mirrored 4TB address space of 4KB sectors (i.e., 1 billion sectors) on an 16TB array, we need 4GB for the primary map (1 billion 4-byte entries), 8GB for the inverse map (2 billion 4-byte entries) and two 4-byte counters. However, a RAM-based solution poses the obvious problem of persistence: how do we recover the state of the Gecko address space from power failures?

One possibility is to store some part of the metadata on an SSD. An obvious candidate is the primary map, which is sufficient to reconstruct both the inverse map and the tail / head counters. Random reads on SSDs are fast enough (at roughly 200 microseconds) to exist comfortably in the critical path of a Gecko read. However, the primary map has very little update locality; a series of Gecko writes can in the worst case be distributed evenly across the entire logical address space. As a result, the metadata SSD is subjected to a workload of random 4-byte writes, which can wear it out very quickly.

Instead, Gecko provides persistence across power failures by storing the inverse map on an SSD, as shown in Figure 3. Each 4KB page on the SSD stores 1024 entries in the physical-to-logical map; we call this a metadata block. Accordingly, the larger log on the address space of the disk array is reflected at much smaller scale (a factor of 1K smaller) on the address space of the SSD. The *i*th 4-byte entry on the SSD is the logical address stored in the *i*th physical sector on the disk array. On a brand-new Gecko deployment, each such 4-byte metadata entry on the SSD is set to the 'blank' value, indicating that no valid data exists at that physical location on the array.

Gecko buffers a small number of metadata pages (in the simplest case, just one page) corresponding to the tail of the log in RAM; accordingly, as first-class writes are issued on the logical address space, these buffered metadata pages are modified in-memory. The metadata pages are flushed to the SSD when all entries in them have been updated, with the important condition that these flushes occur in strict sequential logging order. Correspondingly, Gecko also buffers the metadata pages at the head of the log during GC, which updates metadata entries to point to the 'blank' value. As a result of the flush-in-order condition, at any moment in time the SSD consists of two contiguous segments: one containing 'blank' entries and one with non-'blank' entries. As a result, on recovery from power failure, it is a simple task to reconstruct not only the primary map but also the head and tail counters, since they are simply the beginning and end of the contiguous non-'blank' segment.

The metadata buffering scheme described above avoids small random writes to the SSD due to the perfect update locality of the inverse map. However, it does in-

troduce a window of vulnerability; all buffered metadata is lost on a power failure. A useful property of Gecko's log-structured design is that any such data loss is confined to a recent suffix of the log; in other words, the logical drive supported by Gecko simply reverts to an earlier (but consistent) state. If the application does want to guarantee durability of data, it can issue a 'sync' command to the Gecko block device, which causes Gecko to flush its current metadata page ahead of time to the SSD (and do an overwrite subsequently when the rest of the metadata page is updated). Alternatively, if Gecko is implemented as a hardware controller, battery-backed RAM or supercapacitors can be used to store the metadata pages being actively modified.

Under normal operation, this solution imposes a gentle, sequential workload on the SSD. The SSD only sees two 4KB page writes (one to change the entry from 'blank' to a valid location, and another to change it back during GC) for every 1024 4KB writes to the Gecko array. One of these writes can be avoided if the SSD supports a persistent trim command [16], since metadata blocks at the head can be trimmed instead of changed back to 'blank'. In the example above of a 16TB disk array with a mirrored 4TB address space, an 8GB SSD with 10K erase cycles (which should cost somewhere between $8 and $16 at current flash prices) should be able to support 10K times 8TB of writes, or 80PB of writes.

3.2 Caching

In Gecko the role of caching is multi-fold: to reduce read latencies to data, but also to prevent application reads from interfering with writes (read-write contention). In conventional storage designs, it is difficult to predict which data to cache in order to minimize read-write contention. In contrast, eliminating read-write contention in Gecko is simply a matter of caching the data on the tail drive in the system, thus avoiding any disruption to the write throughput of the array.

To do so, Gecko uses a combination of RAM and an SSD (this can be a separate volume created on the same SSD used for storing metadata, or a separate SSD). When data is first written to a Gecko volume, it is sent to the tail drive and simultaneously cached in RAM. As a result, if the data is read back immediately, it can be served from RAM without disturbing sequentiality of the tail drive. As the tail drive and RAM cache continue to accept new data, older data is evicted from the RAM cache to the SSD cache in simple FIFO order (taking overwrites on the Gecko logical address space into account), and the SSD cache in turn uses an LRU-based eviction policy.

This simple caching scheme also prolongs the lifetime of the SSD cache by coalescing overwrites in the RAM cache. It is partly inspired by the technique of using a hard disk as a write cache for an SSD [23], and similarly extends the lifetime of the SSD by 2X to 8X.

Additionally, Gecko can optionally use RAM and SSD (again, another volume on the same SSD or a different drive) as a read cache for the body of the log, with the goal of improving read performance on the body of the log. In the rest of the paper, we use the term 'SSD cache' to refer to the tail cache, unless explicitly specified otherwise.

3.3 Smarter Cleaning

Thus far, we have described the system as using move-to-tail GC, a simple cleaning scheme where data is moved in strict log order from the head of the log to its tail. While this scheme ensures that GC reads do not interfere with write throughput, GC writes do impact first-class writes to some extent. In particular, GC writes in move-to-tail GC do not disrupt the sequentiality of the tail drive, but instead take up a proportion of the sequential bandwidth of the drive; in the worst case where every element in the log is valid and has to be re-appended, this proportion can be as high as 50%, since every first-class write is accompanied by a single GC write.

To prevent GC writes from interfering with first-class writes, Gecko supports a more sophisticated form of GC called 'compact-in-body'. The key observation in compact-in-body is that any valid entry in the body of the log can be moved to any other position that succeeds it in the log without impacting correctness. Accordingly, instead of moving data from the head to the tail, we move it from the head to empty positions in the body of the log.

The cleaning process for compact-in-body GC is very similar to that of move-to-tail GC. It examines the next physical entry at the head of the log, checks if it is occupied by consulting the inverse physical-to-logical map, and if so, finds a free position in the body of the log between the current head and current tail. It then increments the head counter but leaves the tail counter alone (unless no free positions were found in the body of the log, forcing the update to go to the tail). Finding a free position requires the cleaning process to periodically scan ahead on the inverse map and create a free list of positions. These scans occur on the metadata SSD rather than the disk array and hence do not impact read throughput on the body of the log.

Compact-in-body has the significant benefit compared to move-to-tail that GC activity is now completely independent of first-class writes. It creates space at the head of the log by moving data to the body of the log rather than its tail, and hence does not use up a proportion of the write bandwidth of the tail drive. In addition, it requires no changes to the metadata or caching schemes described above.

However, as described, compact-in-body does have one major disadvantage; it randomizes the workload seen by the metadata SSD, since we are moving data from the head to free positions in the log, which could be ran-

domly distributed. In practice, the difference in write bandwidth of a Gecko chain running move-to-tail GC versus compact-in-body GC is at most a factor of two, since move-to-tail GC uses up 50% of the tail drive's write bandwidth in the worst case whereas compact-in-body does not use any. Accordingly, we provide users the option of using either form of GC, depending on whether they want to maximize write bandwidth or minimize SSD wear.

3.4 Discussion

Chain Length: As mentioned previously, chained logging is based on the premise that the sequential write throughput of a single, uncontended drive is preferable to the overall throughput of multiple, contention-hit drives. This argument obviously does not scale to a large number of drives; beyond a certain array size, the random write throughput of the entire array will exceed the sequential throughput of a single drive. The shorter the length of the chain, the more likely it is that chained logging will outperform conventional RAID-0 over the same number of drives.

However, longer chains have other benefits, such as the improved read throughput that results from having multiple disk heads service the body of the log. Another reason for longer chains is that it allows capacity to be added to the physical log. This capacity can be used to either extend the size of the supported address space, or to lower garbage collection stress on the same address space. In practice, we find that chains of two to four drives provide a balance between write throughput, read throughput and capacity.

Multiple Chains: We expect multiple Gecko chains to be deployed on a single system; for example, a 32-core system with 24 disks might have four mirrored chains of length 3, each serving a set of 8 cores. A single metadata SSD can be shared by all the chains, since the metadata has a simple one-to-one mapping to the physical address space of the entire system. A single cache SSD can be partitioned across chains, with each chain using a 32GB cache.

On a large system with multiple chains, each chain can be extended or shortened on the fly by moving drives to and from other chains, as the occupancy (and consequently, GC demands) of the supported address space and the read/write ratio of the workload change over time. Read-intensive workloads require more disks to be dedicated to the body of the chain.

System Cost: The design described thus far requires: an SSD read cache for the tail, an SSD read cache for the body, a metadata SSD, and a few GB of RAM per chain. Consider an array of 30 512GB drives (15TB in total), organized into 5 mirrored chains of length 3. Based on our experience with Gecko, each such chain requires 2GB of RAM, 32GB of flash for the tail cache, 32GB of flash for the body cache, and 1.5GB of flash for metadata; the total for 5 chains is 10GB RAM and around 340GB of flash. At current RAM and flash prices, this amounts to less than $500, a reasonably small fraction of the total cost for such a system.

Mirroring: As described earlier, a Gecko chain can consist of mirrored drive pairs. Mirroring is very simple to implement; since the drives are paired deterministically and kept perfectly synchronized, none of the Gecko data structures need to be modified. Some benefits of mirroring are obvious, such as fault tolerance against drive failures and higher read throughput. A more subtle point is that Gecko facilitates power saving when used over mirrored drives. Since writes in chained logs only happen at the tail, drives in the body of the log can be powered down as long as one mirror stays awake to serve reads. In a chain consisting of three mirrored pairs, two drives (or a third of the array) can be powered down without affecting data availability. With longer chains, a larger fraction of the array can be powered down.

Additionally, Gecko can potentially perform decoupled GC on mirrors, allowing one drive to serve first-class reads while cleaning the other drive. This complicates the metadata structures maintained by Gecko, both in RAM as well as the metadata SSD, since it needs to now maintain state for each drive separately. Due to the increased complexity of this option, we chose not to explore it further.

Striping: Gecko can also be easily combined with striping, simply by having each drive in the chain be a striped RAID-0 volume. This allows a single Gecko address space to scale to larger numbers of drives. One implication of striping is that the tail drive(s) now have much greater capacity and may require proportionally larger SSD caches to prevent reads from impacting them. Other RAID variants such as RAID-5 and RAID-6 can be layered in similar fashion under Gecko without any change to the system design.

4 Evaluation

We have implemented Gecko as a device driver in Linux that exposes a block device to applications. This device driver implements move-to-tail GC and a simplistic form of persistence involving checkpointing all metadata to an SSD every few minutes. In addition, we also implemented a user-space emulator to test the more involved aspects of Gecko, such as the metadata logging design for persistence described in Section 3.1, compact-in-body GC, and different caching policies. All our experiments were conducted on a system with a 12-core Intel Xeon processor, 24GB RAM, 15 10K RPM drives of 600GB each, and a single 120GB SSD.

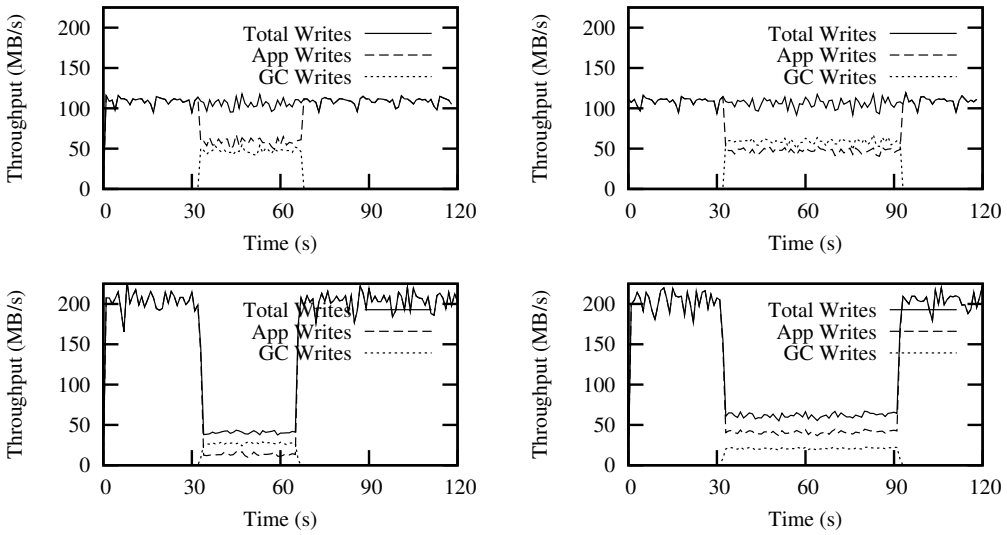

Figure 4: Gecko (Top) offers steady, high application throughput (60MB/s or 15K IOPS) for a random write workload during GC with a 50% trim pattern (Left) and a 0% trim pattern (Right). Log-structured RAID-0 (Bottom) suffers application throughput collapse to 10MB/s for 50% trims (Left) and provides 40MB/s for 0% trims.

Our main baseline for comparison is a conventional log layered over either RAID-0 or RAID-10 (which we call log-structured RAID-0 / RAID-10), comparable respectively to the non-mirrored and mirrored Gecko deployments. For instance, an array of six drives may be configured as a 3-drive Gecko chain, where each drive is mirrored; for this, the comparison point would be a log-structured RAID-10 volume with three stripes, each of which is mirrored. To implement this log-structured RAID design, we treat the entire array as a single RAID-0 or RAID-10 volume and then run a single-drive Gecko chain over it; this ensures that we use identical, optimized code bases for both Gecko and the baseline. When appropriate, we also report numbers on in-place (as opposed to log-structured) RAID-0, though most of our workloads have enough random I/O that in-place RAID-0 only offers a few MB/s and is not competitive.

Our evaluation focuses on three aspects of Gecko. First, we show that a Gecko chain implementing move-to-tail GC is capable at operating at high, stable write throughput even during periods of high GC activity under an adversarial workload, whereas the write throughput of log-structured RAID-0 drops drastically. This validates our claim that Gecko write throughput does not suffer from contention with GC reads. Second, we show that our RAM+SSD caching policies are capable of eliminating almost all first-class reads from the tail drive for a majority of tested workloads, while preserving the lifetime of the SSD cache. Thus, we show that Gecko write throughput does not suffer from contention between application reads. Finally, we play back real traces on a Gecko deployment and show that Gecko offers higher write throughput as well as higher read throughput compared to log-structured RAID-10.

4.1 Write Throughput with GC

To show that Gecko can sustain high write throughput despite concurrent GC, we ran a synthetic workload of random writes from multiple processes over the block address space exposed by the Gecko in-kernel implementation. In this experiment, we used a 2-drive, non-mirrored Gecko chain and a conventional log layered over 2-drive RAID-0. Midway through the workload, we turned on GC for Gecko and measured the resulting drop in total and application throughput. For the log-structured RAID-0, we triggered GC for the same time period as Gecko. Figure 4 (Top) shows Gecko throughput for different trim patterns in the body of the log; e.g., a trim pattern with 50% valid data has half the blocks in the body of the log marked as invalid, while the other half is valid and has to be moved by GC to the tail.

As shown in the figure, Gecko throughput remains high and steady during GC activity, while application throughput drops proportionally to accommodate GC writes. We trigger GC to clear a fixed amount of physical space in the log; as a result, the 50% trim pattern (Top Left) has a GC valley that is approximately half as wide as that of the 0% trim pattern (Top, Right), since it moves exactly half the amount of data. The two different trim patterns on the body of the log do not impact Gecko write throughput in any way, showing that the strategy of decoupling the tail of the log from its body succeeds in shielding write throughput from GC read activity.

In contrast, the log-structured RAID-0 in Figure 4 (Bottom) performs very poorly when GC is turned on for

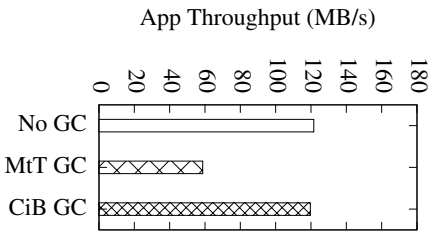

Figure 5: With compact-in-body GC (CiB), a log chain of length 2 achieves 120MB/s application throughput on random writes with concurrent GC on 50% trims.

the 50% trim pattern; throughput collapses drastically to the 10MB/s mark. Counter-intuitively, it performs better for 0% trim pattern; even though more data has to be moved in this pattern, the GC reads to the drive are sequential, causing less disruption to the write throughput of the array. An important point is that Gecko cleans 2X to 3X the physical space compare to log-structured RAID-0 in the same time period: the top Gecko graphs show almost 4GB of log space being reclaimed while the bottom log-structured RAID-0 graphs show reclamation of approximately 1.5 GB of log space in a 40 second (Left) and 60 second (Right) period.

One point to note is that Gecko does suffer from a drop in application throughput, or goodput, due to GC. In the worst case where all data is valid and has to be moved (shown in the top right figure), application throughput can drop by exactly half. This represents a lower bound on application throughput, since in the worst case every new write requires a single GC write to clear up space in the physical log. Accordingly, Gecko application throughput is bounded between 60MB/s (half the sequential bandwidth of a single drive) and 120MB/s (the full sequential bandwidth of a drive), with the exact performance depending on the size of the supported logical address space, as well as the pattern of overwrites observed by it. Not shown in the figure is in-place RAID-0, which provided only a few MB/s under this random writes workload, as expected.

Next, we ran the Gecko emulator in compact-in-body mode as well as move-to-tail mode for a random write workload with a 50% trim pattern. Figure 5 shows that compact-in-body GC allows application writes to proceed at the full sequential speed of the tail drive during GC activity. As discussed previously, this performance benefit comes at the cost of erase cycles on the metadata SSD; accordingly, we do not explore compact-in-body GC further.

4.2 Caching the tail

Having established that Gecko provides high write throughput in the presence of GC activity, we now focus on contention between first-class reads and writes. We show that Gecko can effectively cache data on the tail drive in order to prevent contention between first-class reads and writes. In these experiments, we use block-level traces taken from the SNIA repository [1]; specifically, we use the Microsoft Enterprise, Microsoft Production Server and MSR Cambridge trace sets. Running these traces directly over Gecko is unrealistic, since they were collected on non-virtualized systems. Instead, we run workload combinations by interleaving I/Os from sets of either 4 or 8 traces, to emulate a system running different workloads within separate virtual machines. We play each trace within its own virtual address space and concatenate each of these together to obtain a single logical address space.

Raw Trace	*GB of Writes*
A. DevDivRelease	176.1
B. Exchange	459.6
C. LiveMapsBE	558.2
D. prxy	778.6
E. src1	883.7
F. proj	342.2
G. MSNFS	102.3
H. prn	76.8
I. usr	95.7

Combination 0 – 7: any 8 from {A,...,I}
Combination 8 – 20: any 4 from {A,...,E}

Table 1: Workload Combinations: from 9 raw traces, we can compose 8 8-trace combinations and 13 4-trace combinations that write at least 512GB of data.

To study the effectiveness of Gecko's tail caching, we ran multiple such workload combinations over our user-space Gecko emulator, starting with an empty tail drive. We then measured the hit rate of Gecko's hybrid cache consisting of 2GB of RAM and a 32GB SSD. Recall that new writes in Gecko go to the tail drive and are simultaneously cached in RAM, and subsequently evicted from RAM to the SSD. A cache hit is when data that resides on the tail drive is also found in either RAM or the SSD; conversely, a cache miss occurs when data that resides in the tail drive is not found in RAM or the SSD, necessitating a read from the tail drive. Note that any read to data that does not exist on the tail drive is ignored in this particular experiment, since it will be serviced by the body of the log without causing read-write contention.

To avoid overstating cache hit rates, we needed each workload combination to write at least 512GB (i.e., the size of the tail drive); as we show later, cache hit rates are very high as we start writing to the tail drive, but drop as it fills up. From the 21 SNIA traces, we found 8 8-trace combinations that lasted at least 512GB (which we number 0 to 7), and 13 4-trace combinations that lasted at least 512GB (which we number 8 to 20), for a total of 21 workload combinations of at least 512GB each.

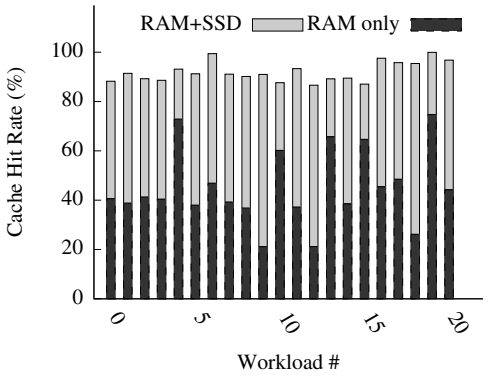

Figure 6: Effectiveness of tail caching on different workload combinations with a 2GB RAM + 32GB SSD cache. The hit rate is over 86% for all 21 combinations, over 90% for 13, and over 95% for 6.

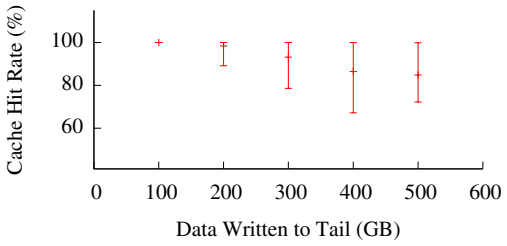

Figure 7: Average, min and max hit rates of tail caching across workload combinations as the tail drive fills up.

These workload combinations used 9 of the 21 raw SNIA traces, as shown in Table 1; the remaining 12 raw traces did not have enough writes to be useful for this caching analysis.

Figure 6 shows cache hit rates – for just the 2GB RAM cache as well as for the combined 2GB+32GB RAM+SSD cache – for these 21 workload combinations, measured over the time that the 512GB tail drive is filled. The hit rate is over 86% for all tested combinations, over 90% for 13 of them, and over 95% for 6 of them. This graph validates a key assumption of Gecko: the tail drive of a chained log can be cached effectively, preventing application reads from disrupting the sequential write throughput of the log.

Next, we measured how the cache hit rate changes over time as the tail drive fills up. Figure 7 shows the average hit rate across the 21 workload combinations for the RAM+SSD cache, in each consecutive 100GB interval on the tail drive (the error bars denote the min and the max across the workload combinations). The hit rate is extremely high for the first 100GB of data, as the total amount of data on the tail drive is not much bigger than the cache. As expected, the hit rate dips as more data is stored on the tail. Note that Figure 6 previously showed the cumulative hit rate over 512GB of writes, whereas this figure shows the hit rate for each 100GB interval separately.

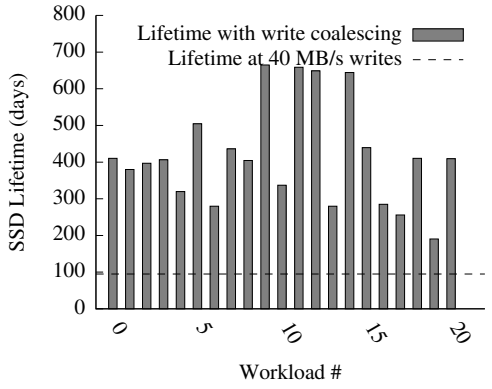

Figure 8: Gecko's hybrid caching scheme for its tail drives increases the lifetime of the SSD read cache by at least 2X for all 21 workload combinations, and by more than 4X for 13 combinations.

We claimed earlier that Gecko's two-tier RAM+SSD caching scheme could prolong the lifetime of the SSD compared to an SSD-only cache by coalescing overwrites in RAM. Following the methodology in [23], we calculate the lifetime of an SSD by assuming a one-to-one ratio between page writes sent to the SSD and erase cycles used per page, and assuming that the SSD supports 10,000 block erase cycles. Under these assumptions, a constant 40MB/s workload will wear out a 32GB SSD in approximately 3 months; accordingly, this would be the lifetime of a conventional SSD-based write or read cache if the system were written to continuously at 40MB/s.

By using a RAM+SSD read cache and coalescing overwrites in RAM, we decrease the number of writes going to the SSD by a factor of 2X to 8X for different workload combinations. In Figure 8, we plot the number of days the SSD lasts with write coalescing, under the assumptions previously stated. For some workload combinations, we are able to stretch out the SSD lifetime to over two years even at this high 40MB/s update rate; for all of them, we at least double the SSD lifetime. A simple linear relationship exists between these numbers and the average data rate of the system; at 20MB/s, for instance, the SSD will last twice as long. Alternatively, we can use larger capacity SSDs to extend the SSD replacement cycle: e.g. with a 64GB SSD, the cycle can double if one uses the first half until it wears out and then uses the other half.

4.3 Gecko Performance for Real Workloads

To show that effective tail-caching results in better performance, we played two 8-trace combinations – specifically, the ones with the highest and lowest cache hit rates – over the Gecko implementation. In this experiment, we

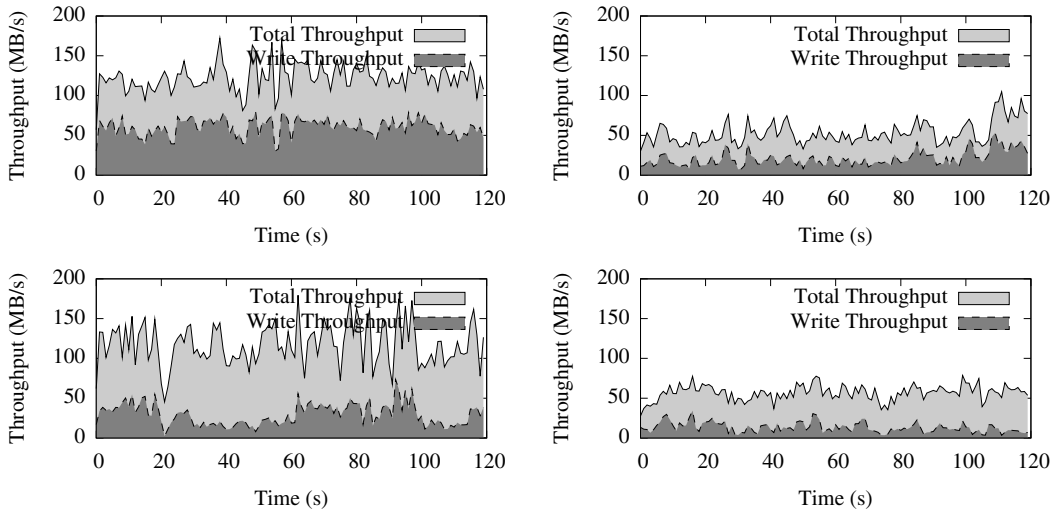

Figure 9: Gecko (Left) offers 2X to 3X higher throughput than log-structured RAID-10 (Right) on a highly cacheable (Top) and less cacheable (Bottom) workload combination for writes as well as reads.

played each trace combination as fast possible, issuing the appropriate I/Os to either the SSD cache or disk. We used a single outstanding I/O queue of size 24 for each trace in the combination, shared by reads and writes.

For Gecko, we used a 3-drive mirrored chain with a 2GB RAM + 32GB SSD tail cache and a separate 32GB SSD cache for the body of the log. For comparison, we used a conventional log over a 6-drive RAID-10 volume with a single unified cache for the entire array, consisting of 2GB RAM and 64GB SSD.

In the experiment, we played the trace combination forward until 200GB of the tail was filled before taking measurements, to ensure that we obtained average caching performance on the tail. Reads on logical addresses that had not yet been written were directed to random locations on the body of the log.

Figure 9 shows the total read plus write throughput of the system as well as just write throughput over a 120 second period. On top we show the highly cacheable workload combination; on bottom we show the less cacheable one. On the left we show Gecko performance, while on the right we show the performance of log-structured RAID-10. No GC activity was triggered concurrently, in order to isolate the impact of first-class reads on performance.

At a basic level, it's clear that Gecko outperforms log-structured RAID-10 by 2X to 3X on both workloads. Gecko offers lower write performance than expected, since write throughput is not pegged at 120MB/s; this is an artefact of our trace playback process, since our fixed-size window of I/Os ends up clogged with the slower reads on the body of the log, preventing new writes from being issued. Surprisingly, Gecko offers much better read performance than log-structured RAID-10, again by a factor of 2X to 3X; in effect, separating reads and writes has a positive effect on reads, which do not have to contend with write traffic anymore. Especially, all fresh reads that are not cached from recent writes contend with writes in both cache and disks for log-structured RAID-10 and this significantly lowers the throughput for both reads and writes. An interesting point is that both workloads are highly cacheable for reads; our classification of these workloads as highly cacheable and less cacheable was based on the cacheability of the tail drive, which does not seem to correlate to the cacheability of the body.

To test the performance under GC, we triggered move-to-tail GC in the same setup as in Figure 9 with 700GB of data pre-filled. Approximately 75% of data was trimmed for both workloads and the average total throughputs of Gecko dropped to 65MB/s and 62MB/s for highly and less cacheable workloads respectively due to contention between first-class reads and GC reads. However, Gecko still outperformed log-structured RAID-10 performing GC by over 2X to 3X. The average application throughputs of Gecko were 33MB/s and 27MB/s whereas those of log-structured RAID-10 were 13MB/s and 12MB/s for the respective workloads.

Finally, we plot the impact of chain length on throughput in Figure 10. We run the highly cacheable workload from the previous experiments on Gecko and log-structured RAID-0, measuring read and write throughput while increasing the number of drives used without GC. In Gecko, more drives in the array translates into more drives in the body of the chain, while for log-structured RAID-0 it provides more disks to stripe over. As the graph shows, a single Gecko chain outperforms log-structured RAID-0 for both reads and writes even on a 7-disk array. Essentially, two key principles in the

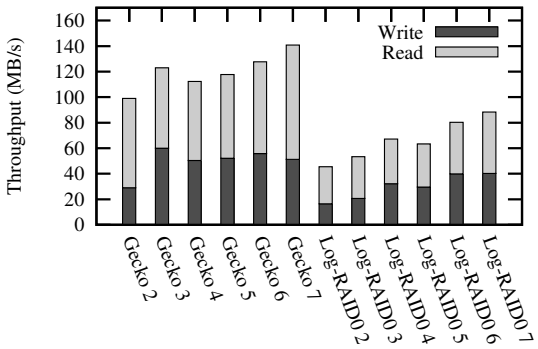

Figure 10: A Gecko chain outperforms log-structured RAID-0 even on 7 drives: one uncontended disk for writes is better than many contention-prone disks.

Gecko design continue to hold even for long chains: a single uncontended disk arm is better for write performance than multiple contended disk arms; and segregating reads from writes enables better read performance.

5 Related Work

Log-structured storage has a long and interesting history, starting with the original LFS paper [18]. Much of the work on LFS in the 90s focused on its shortcomings related to garbage collection [20, 13, 21]. Other work, such as Zebra [11], extended LFS-like designs to distributed storage systems. Attempts to distribute logs focused entirely on striping logs over multiple drives, as opposed to the chained log design we present.

More recently, contention in data centers has received increasing attention. Lithium [10] uses a single on-disk log structure to support multiple VMs, much as Gecko does, but layers this log conventionally over RAID and does not offer any new solutions to the problem of read-write contention. The authors do make two relevant points: first, that replicated workloads are even more likely to be write-dominated, and second, that the inability of log-structured designs to efficiently service large, sequential reads is unlikely to matter in virtualized settings where such reads are rare due to cross-VM interference. Parallax [15] supports large numbers of virtual disks over a shared block device, but focuses on features such as frequent snapshots rather than performance under contention. PARDA [7] is a system that provides fair sharing of a storage array across multiple VMs, but does not focus as we do on improving aggregate throughput under contention.

Log-structured designs have made a strong comeback in recent years. One reason is the emergence of flash, which requires a log-structured design to minimize wear-out. Not only do individual SSDs layer an address space over a log, but filesystems designed to run over SSDs are often log-structured to minimize the stress on the SSD's internal mapping mechanisms [2]. New log-structured designs have emerged as flash becomes mainstream; for instance, [3] uses an off-path sequencer to implement a distributed, shared log over a flash cluster. Another reason for the return of log-structured designs is the increased prevalence of geo-distributed systems, where the intrinsic ordering properties of logs provide consistency-related benefits [24].

Gecko is inspired heavily by a long line of block-level storage designs, starting with RAID [17]. Such designs typically introduced a layer of indirection at the block level for higher reliability and better performance; for instance, the Logical Disk [6] implemented a log-structured design at the block level for better performance. Log-structured arrays [14] layered a log-structured design over a RAID-5 array. HP AutoRAID [25] switched dynamically between RAID-1 and RAID-5 for hot and cold data, respectively. Petal [12] extended this design to a distributed setting, maintaining an indirection map that could support arbitrary mappings between a logical address space and physical disks. Such systems typically used battery-backed RAM for persisting block-level metadata [19, 14]. While Gecko is similar to these systems in philosophy, it benefits from the ready availability of commodity flash for achieving persistence, but consequently has to work around the wear-related limitations of flash. Finally, we explored this design previously in a workshop paper, without a working system or evaluation [22].

6 Conclusion

A number of factors herald a second coming for log-structured storage designs: the emergence of cloud computing, the prevalence of many-core machines, and the availability of flash-based read caches. Log-structured designs have the potential to be a panacea for storage contention in the cloud; however, they continue to be plagued by the cleaning-related performance issues that held back widespread deployment in the 90s. Gecko attempts to solve this long-standing problem by separating the tail of the log from its body, thus isolating cleaning activity completely from application writes. A dedicated cache for the tail drive prevents first-class reads from interfering with writes. Using these mechanisms, Gecko offers the benefits of a log-structured design without its drawbacks, presenting system designers with a contention-oblivious storage option in the cloud.

Acknowledgements

This work was partially funded and supported by a NetApp Faculty Fellowship and IBM Faculty Award received by Hakim Weatherspoon, DARPA (D11AP00266) and NSF (1053757, 0424422, 1040689, 1151268, 1047540). We would like to thank our shepherd, Kiran-Kumar Muniswamy-Reddy, and the anonymous reviewers for their comments.

References

[1] SNIA IOTTA Repository. http://iotta.snia.org/.

[2] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J., MANASSE, M., AND PANIGRAHY, R. Design Tradeoffs for SSD Performance. In *Proceedings of USENIX Annual Technical Conference* (2008), pp. 57–70.

[3] BALAKRISHNAN, M., MALKHI, D., PRABHAKARAN, V., WOBBER, T., WEI, M., AND DAVIS, J. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation* (2012), pp. 1–14.

[4] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., XU, Y., SRIVASTAV, S., WU, J., SIMITCI, H., ET AL. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of ACM Symposium on Operating Systems Principles* (2011), pp. 143–157.

[5] CHAO, C., ENGLISH, R., JACOBSON, D., STEPANOV, A., AND WILKES, J. Mime: A High Performance Parallel Storage Device with Strong Recovery Guarantees. Tech. rep., HP Labs Technical Report HPL-CSP-92-9, 1992.

[6] DE JONGE, W., KAASHOEK, M., AND HSIEH, W. The Logical Disk: A New Approach to Improving File Systems. *ACM SIGOPS Operating Systems Review 27*, 5 (1993), 15–28.

[7] GULATI, A., AHMAD, I., AND WALDSPURGER, C. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *Proccedings of USENIX Conference on File and Storage Technologies* (2009), pp. 85–98.

[8] GULATI, A., KUMAR, C., AND AHMAD, I. Storage Workload Characterization and Consolidation in Virtualized Environments. In *Workshop on Virtualization Performance: Analysis, Characterization, and Tools* (2009).

[9] GULATI, A., SHANMUGANATHAN, G., AHMAD, I., WALDSPURGER, C., AND UYSAL, M. Pesto: Online Storage Performance Management in Virtualized Datacenters. In *Proceedings of the ACM Symposium on Cloud Computing* (2011), p. 19.

[10] HANSEN, J., AND JUL, E. Lithium: Virtual Machine Storage for the Cloud. In *Proceedings of the ACM Symposium on Cloud Computing* (2010), pp. 15–26.

[11] HARTMAN, J., AND OUSTERHOUT, J. The Zebra Striped Network File System. *ACM Transactions on Computer Systems 13*, 3 (1995), 274–310.

[12] LEE, E., AND THEKKATH, C. Petal: Distributed Virtual Disks. *ACM SIGOPS Operating Systems Review 30*, 5 (1996), 84–92.

[13] MATTHEWS, J., ROSELLI, D., COSTELLO, A., WANG, R., AND ANDERSON, T. Improving the Performance of Log-Structured File Systems with Adaptive Methods. In *Proceedings of the ACM Symposium on Operating System Principles* (1997), pp. 238–251.

[14] MENON, J. A Performance Comparison of RAID-5 and Log-Structured Arrays. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing* (1995), pp. 167–178.

[15] MEYER, D., AGGARWAL, G., CULLY, B., LEFEBVRE, G., FEELEY, M., HUTCHINSON, N., AND WARFIELD, A. Parallax: Virtual Disks for Virtual Machines. In *Proceedings of European Conference on Computer Systems* (2008), pp. 41–54.

[16] NELLANS, D., ZAPPE, M., AXBOE, J., AND FLYNN, D. ptrim ()+ exists (): Exposing New FTL Primitives to Applications. In *Annual Non-Volatile Memories Workshop* (2011).

[17] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1988), pp. 109–116.

[18] ROSENBLUM, M., AND OUSTERHOUT, J. K. The Design and Implementation of a Log-Structured File System. *ACM Transaction on Computer Systems 10* (1992), 26–52.

[19] SAVAGE, S., AND WILKES, J. AFRAID: A Frequently Redundant Array of Independent Disks. In *Proceedings of USENIX Annual Technical Conference* (1996), pp. 3–3.

[20] SELTZER, M., BOSTIC, K., MCKUSICK, M., AND STAELIN, C. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the USENIX Winter 1993 Conference* (1993), pp. 3–3.

[21] SELTZER, M., SMITH, K., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File System Logging Versus Clustering: A Performance Comparison. In *Proceedings of USENIX Annual Technical Conference* (1995), pp. 21–21.

[22] SHIN, J.-Y., BALAKRISHNAN, M., GANESH, L., MARIAN, T., AND WEATHERSPOON, H. Gecko: A Contension-Oblivious Design for Cloud Storage. In *USENIX Workshop on Hot Topics in Storage and File Systems* (2012).

[23] SOUNDARARAJAN, G., PRABHAKARAN, V., BALAKRISHNAN, M., AND WOBBER, T. Extending SSD Lifetimes with Disk-Based Write Caches. In *Proceedings of USENIX Conference on File and Storage Technologies* (2010), pp. 8–8.

[24] WEATHERSPOON, H., EATON, P., CHUN, B., AND KUBIATOWICZ, J. Antiquity: Exploiting a Secure Log for Wide-Area Distributed Storage. *ACM SIGOPS Operating Systems Review 41*, 3 (2007), 371–384.

[25] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems 14*, 1 (1996), 108–136.