Gecko: A Contention-Oblivious Design for Cloud Storage

Ji-Yong Shin[*], Mahesh Balakrishnan[‡], Lakshmi Ganesh[§], Tudor Marian[†], Hakim Weatherspoon[*]
[*] *Cornell University,* [‡] *Microsoft Research,* [§] *UT Austin,* [†] *Google*

Abstract

Disk contention is a fact of life in modern data centers, with multiple applications sharing the storage resources of a single physical machine. Log-structured storage designs are ideally suited for such high-contention settings, but historically have suffered from performance problems due to cleaning overheads. In this paper, we introduce Gecko, a novel design for storage arrays where a single log structure is distributed across a chain of drives, physically separating the tail of the log (where writes occur) from its body[1]. This design provides the benefits of logging – fast, sequential writes for any number of contending applications – while eliminating the disruptive effect of log cleaning activity on application I/O.

1 Introduction

Modern data centers are heavily virtualized, with the compute and storage resources of each physical server multiplexed over a large number of applications. Two trends point towards increased virtualization. The first is the emergence of cloud computing, where cloud providers routinely assign different cores on a single machine to different tenants. 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 a single physical machine.

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 [2]. 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. 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.

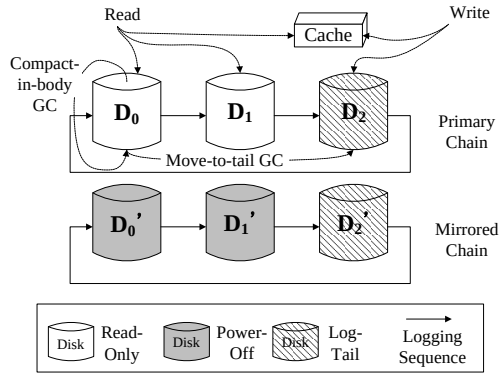

Figure 1: Chained logging: all writes go to the tail drive, while reads take place mostly from the body of the chain or a cache. Mirrors in the body can be powered down.

One solution to minimize interference involves careful placement of applications on machines [2]. 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, etc. A second 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 acceptable for many applications.

Log-structured designs for storage can alleviate contention in such settings. For instance, a log-structured filesystem (LFS) [6] can support sequential or random write streams from multiple applications at the full sequential speed of the underlying disk array. Unfortunately, the Achilles' Heel of LFS is cleaning or garbage collection (GC) [7, 4]; specifically, the random reads introduced by GC to move data from the body of the log to its tail often interfere with first-class writes by the application, negating any improvement in write throughput.

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

[1] Geckos are known to drop their tails in stressful situations and run very fast, much like our system.

– 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. Note that 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, or even striped (e.g. five Gecko chains can constitute stripe groups for RAID-0, 5, or 6 configuration.).

Chained logging offers a trade-off: it reduces the maximum, contention-free write throughput of a disk array, and instead offers stable, predictable write performance in the face of contention. Since only a subset of the drives in the array (specifically, those constituting the tail of the log) receive writes at any given time, the contention-free throughput of the array is less than in comparable designs such as RAID-0. However, chained logging eliminates performance degradation due to write-write contention between applications; all writes occur sequentially at the drives constituting the tail of the log. Critically, chained logging can ensure that GC activity minimally impacts application throughput; GC reads always go to the body of the log, which exists on a different subset of drives from the tail, and GC writes either sequentially go to the body or to the tail depending on the GC strategy.

To tackle read-write contention, in which reads from one application can contend with writes from some other application, Gecko leverages the unique structure of the logging chain. On its own, chained logging ensures that only a fraction of reads in the system can interfere with write throughput; namely, only reads that access the tail drive. For these reads, Gecko places a flash-based cache in front of the disk array and preferentially caches data from the tail drive of the chain. Our preliminary investigations suggest that a 34 gigabyte cache can absorb over 86% of reads from mixes of real workloads to tail drives (See Section 4). Further, Gecko uses a cache to handle read-read contention between applications; we use a multi-level-cell (MLC) flash-based read cache, because it is cheap with sufficient capacity, but can handle (random) reads efficiently.

When composed into larger arrays with mirroring or striping, chained logging provides other benefits. Since disks in the body of the log chain do not receive any first-class writes, their mirrors can be switched off to save power. This lowers the read throughput of the log, but does not compromise fault tolerance. Alternatively, Gecko can keep a mirror powered on and serve first-class reads from it while the other mirror is being cleaned for garbage collection. As a result, at any given point, at least one mirror is satisfying reads without interference from cleaning activity.

2 Disk Contention in the Data Center

Our focus is on settings where multiple applications share a 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 maintains disk scheduling policies and processes I/O independently as if it resides on its own physical disk. In reality, virtual disks are 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 often the case that different virtual disks 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 the disk can disrupt the throughput of every other application running on a machine [2]. As machines come packed with increasing numbers of cores – and as cloud providers cram more tenants on a single physical box – it becomes more 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 low even if every application on the system is 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 [3].

To illustrate these problems, we ran a set of simple experiments on an 8-core machine with 4 disks configured as a RAID-0 array. In the experiments, 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).

Figure 2 (Left) shows measurements of system throughput with increasing numbers of sequential writers and no random writers. For all levels of layering, as we increase the number of sequential writers, aggregate throughput drops substantially (by more than 50% in the case of VMs writing to a filesystem) but does not col-

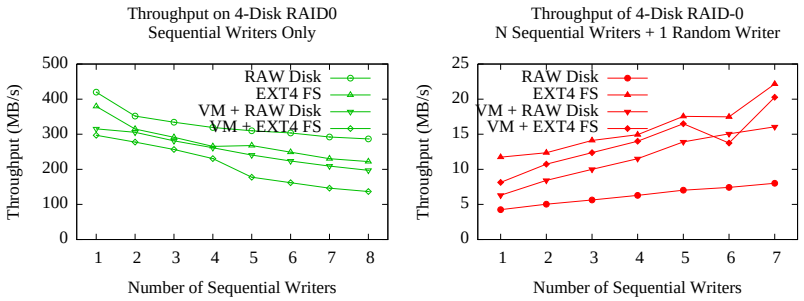

Figure 2: Throughput of 4-disk RAID-0 storage under N sequential or N sequential + 1 random writers.

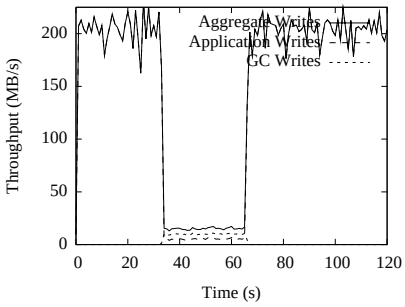

Figure 3: LFS throughput for a 2-disk RAID-0 array collapses when GC kicks in at around the 30s mark.

lapse. As might be expected, more layering results in lower throughput across the board. The sequential writes in this experiment are 256KB each.

Figure 2 (Right) shows the result of a similar experiment; this time, we introduce a single random writer issuing 4KB writes. For any number of sequential writers and any degree of layering, throughput collapses by more than an order of magnitude when the random writer is introduced. Notice the difference in scale for the left and right figures. Interestingly, added layering improves throughput in the presence of a random writer; we surmise that this is due to scheduling intelligence in these layers that delays random I/O to improve sequentiality.

3 The Return of LFS and Limitations

Log-structured filesystems were introduced in the mid-90s on the key assumption that cheap RAM would make it easy to cache hot data, resulting in write-dominated workloads. Today, a similar argument can be made in the context of flash. Flash drives are now inexpensive enough to provide relatively large read caches for disk arrays that can filter out a large fraction of reads. There's still uncertainty over the future role of flash as a disk replacement [1], with flash manufacturers walking a tightrope between reliability and density (and consequently, cost). However, MLC flash enables fast, large and long-lasting read caches; since data in the cache is not expected to be durable, the reduced reliability of MLC flash over time is not a barrier to deployment. MLC-based flash drives are already seeing widespread deployment within data centers in caching roles. As a result, we expect data center disk workloads to be increasingly write-dominated and for LFS to make a comeback.

The other important assumption of LFS – that media supports fast sequential I/O and slow random I/O – continues to hold nearly two decades later. As drives have become larger, modern systems are increasingly starved for I/Os per second (IOPS); a 1 TB drive typically offers a few hundred IOPS in random write mode, placing an immense reservoir of storage behind a tiny funnel.

As we suggest in this paper (and others have noted before [3]), log-structured designs are ideal for reducing disk contention in virtualized settings. However, the main criticism of LFS in the context of disk-based systems concerns the overhead of cleaning the log [7, 4, 8]. Unless the size of the log is significantly greater than the supported address space, log-structured storage systems can experience severe interference between reads issued to the body of the log for GC and first-class application writes issued to the tail of the log. Our experimental result in Figure 3 shows how severely GC can affect the application throughput in LFS with a 2-disk RAID-0 setup. We pre-filled the disks, used a 4KB random write workload, and triggered GC 30 seconds into the experiment. An aggregate throughput of over 200 MB/s suddenly collapses to 10 MB/s when GC is invoked.

4 Gecko Design

At the heart of Gecko is the idea of chained logging, in which a single logical log structure is chained across multiple drives such that the tail of the log and its body are on different drives. As the body of the log becomes increasingly fragmented with free space due to overwrites on the supported address space, cleaning it involves reading the occupied fragments from the body of the log and re-appending them. Chained logging ensures that this activity does not significantly disrupt the write bandwidth of the log; the reads issued while cleaning occur away from the tail and hence do not interfere at all with application writes, while the writes issued by cleaning are perfectly sequential and occupy a fraction or none of the total write bandwidth without disrupting the sequentiality of the write workload seen by the array.

We first discuss the simplest instantiation of chained logging, and then progressively describe more sophisticated versions. To support a virtualized environment, Gecko should be implemented below the guest VM layer. For portability, Gecko implements a block device exposing a linear address space to the host OS (e.g. OS hosting KVM) or the hypervisor (e.g. dom0 of Xen). To imple-

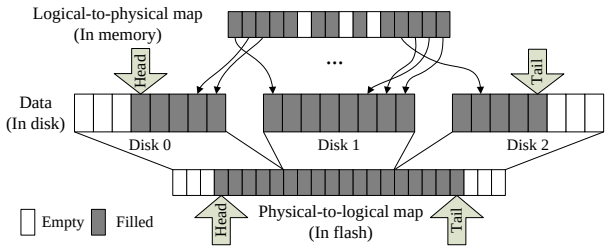

Figure 4: Gecko metadata: a linear logical-to-physical map is in memory and the inverse map is stored in flash for persistence.

ment a chained log, Gecko maintains a metadata structure mapping logical sectors on the linear address space to physical sectors on a chain of disks. It also maintains two counters — one for the tail of the log and one for the head. In addition, it maintains an inverse map to find the logical sector that a physical location stores; a special value is used to indicate that the location is empty. Note that the total amount of the metadata required is not excessively high; with 4KB sectors, an 8 TB array requires only around 8 GB of metadata for both the primary and the inverse map, which can fit comfortably on a small flash drive.

In the simplest case, garbage collection of the log is performed in strict log order; i.e., a cleaning process examines the next entry at the head of the log, checks whether the logical-to-physical map and the inverse map point to each other to test validity of the entry, and either discards it or re-appends it to the tail of the log, updating one or both counters in the process. The inverse map is updated to the special value after garbage collection. When applications issue writes over the linear address space, the new data goes to the tail of the log and the metadata is updated: the logical-to-physical map now points the modified logical sectors to the new entries in the physical log, the inverse map is updated accordingly, and the tail counter moves forward.

Gecko metadata needs to be persisted for recovery from power failures; for this Gecko uses flash. Rather than logical-to-physical map, which has little update locality and can shorten the flash lifetime, Gecko stores the inverse map (Figure 4). Gecko keeps a small fraction of the inverse map of the tail drive in memory and flushes 4KB pages, which store 1024 4-byte entries, to the flash in strict order whenever the page becomes fully written. On recovery from power failure, the head/tail counters can be learned from the beginning and end of special empty values in the inverse map (i.e. physical-to-logical map). Then, reading the map from the tail to the head can reconstruct the logical-to-physical map.

The simple system described thus far provides the main benefit of log chains – logging without cleaning interference – but suffers from other problems. It does not offer tolerance to disk failures and restricts the write throughput of an entire array to a single drive. While cleaning 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, reads to recently written data will go to the tail disk and disrupt the write throughput of the system. Below, we discuss extensions to address these concerns.

Smarter Cleaning: To prevent the write activity of cleaning from interfering with first-class application writes, Gecko can compact data within a single drive in order to free up spare capacity. In this case, the cleaning process periodically compacts data in the body of the log, moving around data on the same disk that it resides on as opposed to moving it to the tail of the log chain. The benefit of this approach is that the tail of the log chain is now completely isolated from all cleaning activity, including both reads and writes. However, this requires extra metadata to be maintained; Gecko now needs to maintain head and tail counters for each disk and store which drive contains the global head and tail in the flash for persistence. We call this 'compact-in-body' GC and the simpler form 'move-to-tail' GC (Figure 1).

Mirroring: To provide tolerance to disk failures, Gecko mirrors each drive in the chain. This does not require additional metadata to be maintained, since drives are deterministically paired and synchronously updated together. Combining mirroring with chained logging creates new opportunities. For instance, the chained log ensures that drives in the body of the log do not receive first-class writes; accordingly, one of the mirrors can be powered off if the read load does not require more than a single drive to service it. For a mirrored chain of length 3 with six drives, two drives can be powered down at any given point, resulting in a power savings of 33%. Alternatively, Gecko can implement decoupled cleaning across mirrors, cleaning one mirror using move-to-tail GC while servicing application reads from the other. This prevents cleaning activity from interfering with application reads on the body of the chain.

Striping: In a Gecko chain, the write throughput of an array is limited by a single drive (or pair of mirrored drives). In essence, the argument is that an uncontended single drive offers superior throughput compared to contention-prone access to an array of drives (configured in RAID-0, for instance). Beyond a certain array size / chain length, the contended random write throughput of a full array may exceed the sequential write throughput of a single drive, at which point chained logging ceases to be the better option. As a result, chained logging does not scale on its own to large arrays of tens of drives. To scale to larger arrays, data can be striped across multiple Gecko chains using RAID (striping) configurations (e.g. RAID 0, 5, and 6).

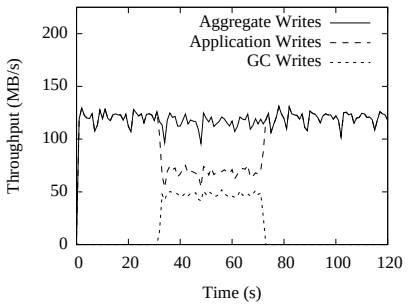

Figure 5: Gecko in-kernel: on a workload of 4KB random writes, a log chain of length 2 achieves 120 MB/s. When move-to-tail GC kicks in about 30s into the experiment, the aggregate throughput is undisturbed.

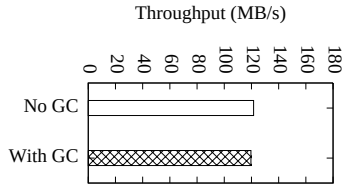

Figure 6: Gecko emulator: with compact-in-body GC, a log chain of length 2 achieves 120 MB/s application throughput on random writes regardless of GC activity.

Caching: Gecko preferentially caches data on its tail drives to prevent application reads from contending with write throughput. To validate the practicality of this approach, we examined mixes of 8 enterprise server block-level traces taken from a 2009 Microsoft study [5]. For all cases, we found that over 86% of reads on 500GB tail drives could be eliminated with a 34GB cache. We believe that MLC flash can serve as effective media for read caches; a 40 GB Intel MLC SSD that costs $80 can cache 2.5% of a 1.6 TB address space.

5 Current Status

Gecko is currently implemented as an in-kernel block device driver that exposes a logical volume to applications. We also have a user-space emulator that we use to rapidly evaluate different policies before implementing them in the in-kernel version. Our in-kernel version currently implements simple move-to-tail cleaning, as described above, moving data from the body to the tail of the log.

Figure 5 shows preliminary performance numbers for our in-kernel Gecko implementation while running synthetic 4KB random write workloads. The experiment is run on a 2-drive log chain. At the starting point of the experiment, the first drive in the chain has already been filled and constitute the body of the chain, and new writes go to the second drive. In the first part of the graph, a synthetic workload generator issues random 4KB writes to the address space and no GC activity is present; accordingly, the system maximizes the bandwidth of the tail drive at 120 MB/s. Then, GC activity kicks in and begins moving data from the body of the log to the tail drive; this causes application throughput to drop to 70 MB/s, with GC writes taking up 50 MB/s. Compared to how LFS performed in Figure 3, Gecko handles GC very well due to GC reads taking place in the body.

In the user-space emulator, we implemented the more sophisticated compact-in-body GC. Figure 6 shows that application write throughput is completely unaffected by GC activity under the same experimental setup as the in-kernel version, since all cleaning reads and writes go to the body of the log.

6 Conclusion

A number of factors herald a second coming for log-structured storage designs, including the emergence of cloud computing, the prevalence of many-core machines, and the availability of high-capacity flash-based read caches. However, log-structured designs are still 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.

Acknowledgments

This work was partially funded and supported by a NetApp Faculty Fellowship, IBM Faculty Award, and NSF CAREER Award received by Hakim Weatherspoon. We would like to thank Johannes Gehrke, our shepherd, Himabindu Pucha, and the anonymous reviewers for their comments.

References

[1] GRUPP, L., DAVIS, J., AND SWANSON, S. The bleak future of nand flash memory. In FAST (2012).

[2] GULATI, A., KUMAR, C., AND AHMAD, I. Storage workload characterization and consolidation in virtualized environments. In VPACT (2009).

[3] HANSEN, J., AND JUL, E. Lithium: Virtual machine storage for the cloud. In SOCC (2010).

[4] MATTHEWS, J., ROSELLI, D., COSTELLO, A., WANG, R., AND ANDERSON, T. Improving the performance of log-structured file systems with adaptive methods. In SOSP (1997).

[5] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating server storage to SSDs: Analysis of tradeoffs. In Eurosys (2009).

[6] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. ACM Trans. on Comp. Sys. 10 (Feb 1992), 26–52.

[7] SELTZER, M., SMITH, K., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: A performance comparison. In USENIX ATC (1995).

[8] WANG, J., AND HU, Y. WOLF - A novel reordering write buffer to boost the performance of log-structured file systems. In FAST (2002).