Towards Fair Sharing of Block Storage in a Multi-tenant Cloud

Xing Lin[1], Yun Mao[2], Feifei Li[1], and Robert Ricci[1]

[1]University of Utah, School of Computing
[2]AT&T Labs - Research

Abstract

A common problem with disk-based cloud storage services is that performance can vary greatly and become highly unpredictable in a multi-tenant environment. A fundamental reason is the interference between workloads co-located on the same physical disk. We observe that different IO patterns interfere with each other significantly, which makes the performance of different types of workloads unpredictable when they are executed concurrently. Unpredictability implies that users may not get a fair share of the system resources from the cloud services they are using. At the same time, replication is commonly used in cloud storage for high reliability. Connecting these two facts, we propose a cloud storage system designed to minimize workload interference without increasing storage costs or sacrificing the overall system throughput. Our design leverages log-structured disk layout, chain replication and a workload-based replica selection strategy to minimize interference, striking a balance between performance and fairness. Our initial results suggest that this approach is a promising way to improve the performance and predictability of cloud storage.

1 Introduction

Infrastructure-as-a-Service (IaaS) cloud computing services offer virtual machines (VMs) that provide elastic computing, storage and network resources. Exemplified by Amazon EC2, IaaS clouds are attractive because they are cost-effective and simple to manage.

There are several types of storage service abstractions in the cloud, including object stores (e.g. Amazon S3), block stores (e.g. Amazon EBS), and databases (e.g. Amazon RDS and Microsoft SQL Azure). Among these options, block-level storage provides the most flexibility. Because a block device is attached as a conventional disk to a VM, applications do not need to be modified to "port" them to the cloud. A block device can be mapped to a partition of a local attached hard drive, a logical volume from a Storage Area Network (SAN), or a customized driver that leverages a storage cluster. For example, Amazon EBS not only provides block storage, but also offers high reliability by performing replication in the storage cluster.

However, these cloud-based storage services introduce a multi-tenant environment and the performance experienced by the end-users in such an environment varies, sometimes more than an order of magnitude, compared with a dedicated cluster [2, 18]. For example, Shripad and Radu pointed out that at different times of day, the performance of Amazon EBS and S3 also varied significantly [14]. Such performance fluctuations are a natural consequence of sharing servers, networks, and storage among many different users. Workloads from different tenants compete for shared resources. For block storage, when two or more tenants share the same physical disk, they compete for the disk head position for I/O accesses. For instance, random workloads from one tenant can destructively interfere with sequential workloads of another tenant [7], and reads may conflict with writes [1]. Such interference makes the performance experienced by applications highly unpredictable. Attackers may also use the performance anomaly as a covert channel between VMs to perform co-residence checks [16].

Ideally, when there are n similar workloads sharing the storage system, the worst-case slowdown seen by each workload should be a factor of n compared to the performance obtained when running the workload in isolation. However, the unpredictability problem makes this almost impossible: as we will show in Section 2, when different types of workloads are randomly mixed on the same physical disk, the result is an unfair distribution of system resources among tenants.

This paper focuses on improving the performance predictability of block storage in clouds, which naturally leads to a storage service that provides a fair share of system resources in a multi-tenant environment. To demonstrate the problem, we first present some observations and provide several design implications in Section 2. We then present the proposed architecture of our block storage system for the cloud, the *FAST* (*F*air *A*ssignment for *S*torage *T*enants) system. *FAST* provides a block-level replicated storage service for reliability, while aiming to minimize the interference between different tenants. By default, each block device is replicated to three copies in the storage cluster, using chain replication [20] to ensure reliability and durability. *FAST* uses different disk layouts on the replicas. In particular, we adopt *log-structured storage* [17, 10] in one of the replicas. This converts all write operations, whether they are sequential or random from tenant's perspective, into high throughput sequential IOs to reduce interference and improve write performance. We use the conventional layout with *buffered-write storage* in the other replicas. *FAST* intelligently redirects read operations to avoid co-locating random and sequential reads and thus the unpredictable interference between these types of workloads. Finally, we present initial simulation results that validate our hypothesis that *FAST*'s design achieves a fair share of the storage service among users in a multi-tenant cloud environment without sacrificing the overall system throughput.

2 Observations and Design Implications

The design of the FAST system is motivated by our observations of undesirable interference between different types of workloads when they are concurrently executed on the same physical disk. We use the *FIO* tool [3] to generate workloads directly at the block storage level. For higher-level experiments, we use the TPC-H benchmark [19] to simulate real-world application scenarios. Specifically, our objective is to evaluate the performance interference between random and sequential workloads, and between reads and writes.

2.1 Micro-benchmark with FIO

To investigate the performance interference between workloads, we looked into all types of workloads: random read (RR), sequential read (SR), random write (RW), and sequential write (SW). Each workload is set to run for 120 seconds. The block size is set to be 4 KB and we used direct I/O to bypass operating system caches and examine disk behavior directly. The I/O queue depth is set to be 32. We measure random workloads in terms of I/O operations per second (IOPS), and use throughput to measure sequential workloads.[1] We used a Seagate Cheetah 10,000 RPM 146 GB SCSI disk for our tests. The Product ID is ST3146707LC. The SCSI storage controller in use is LSI Logic/Symbios Logic 53c1030 PCI-X Fusion-MPT Dual Ultra320 SCSI. Each workload writes to a different 10 GB physical region of the disk.

Co-locating similar workloads. Our first set of experiments puts workloads of the same type on the same disk, varying the number of workloads from one to eight. We present the results for random read and sequential write workloads in Figure 1; trends for the other two types (not shown) are similar. As we can see, fair share in performance (hence, the system resources) is preserved: for each experiment, all workloads see similar IOPS or throughput.[2]

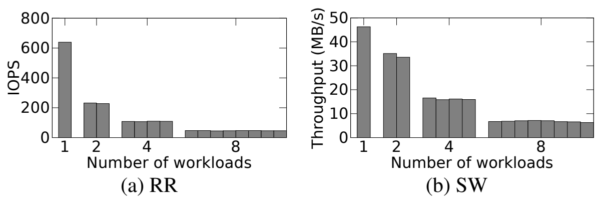


Figure 1: Co-locating the same type of workloads.

➤ *Observation 1*: When co-locating the same type of workloads, each workload gets a fair share in performance and system resources.

Co-locating different types of workloads. The destructive interference from co-locating different types of workloads are presented in Figure 2. Figure 2(a) shows the IOPS achieved by an RR workload (denoted as the *base workload*) co-located with one instance of the four types of workloads. The ideal case for predictability and fair-sharing would be for the base workload to achieve approximately 50% of single-tenant performance. The '1/2' bar shows this ideal case for reference. Figures 2(b), (c) and (d) represent the same results for similar experiments when we change the base workload from RR to SR, RW, and SW respectively. These figures reveal several interesting findings.

➤ *Observation 2*: A random write workload is destructive for all other types of workloads: for all base workloads, the RW bar is the lowest. This holds even for the RW workload itself.

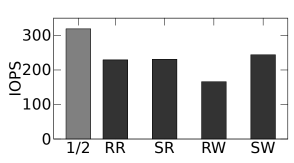
➤ *Observation 3*: Sequential write workloads are seriously impacted by workloads of a different type. The SW bar in Figure 2(d) is the highest, indicating that a sequential workload gets its maximum performance only when it co-locates with another workload of the same type. In fact, the SW bars in all figures are the highest, i.e. all workloads increase their performance by unfairly "stealing" performance from the SW workload.

➤ *Observation 4*: When co-locating two SW workloads, the aggregated throughput is actually larger than the throughput when running a single such workload alone. We further verified that this holds for up to 8 concurrent SW workloads (not shown).

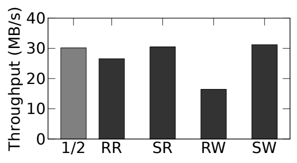
➤ *Observation 5*: It is not worth co-locating the two types of read workloads: RR and SR. Doing so brings

[1]For random workloads, IOPS tells the seek interval directly while for sequential workloads, the throughput is not sensitive to buffer size.

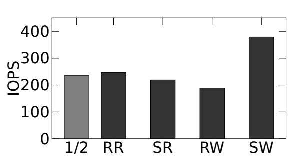
[2]A minor exception is that for all types of workloads, when we ran two workloads of the same type concurrently, one workload always gets a slightly worse performance; we believe this to be an artifact of disk geometry.



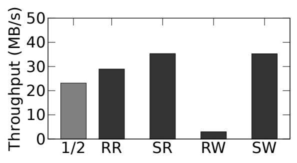
(a) RR



(b) SR



(c) RW



(d) SW

Figure 2: Interference between different types of workloads

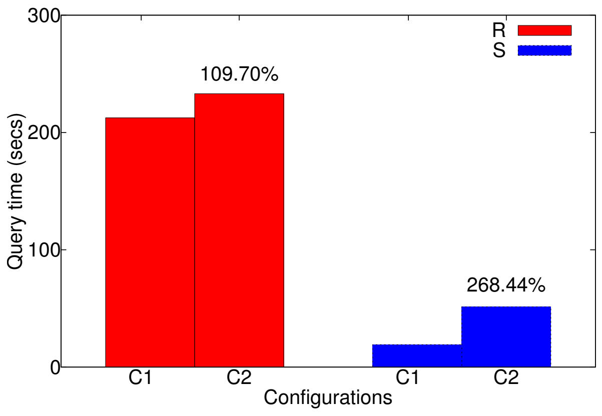


Figure 3: Performance interference in TPC-H.

little improvement in IOPS for RR workloads, but the drop in throughput for SR workloads is more significant. This drop is shown by the difference between RR and SR bars in Figure 2(b).

2.2 Real-world applications

Previous results have demonstrated the interference between workloads at the block storage level. Such interference can be propagated to the application layer. To measure this effect, we used the 21 queries provided with TPC-H as a random workload[3], and 9 queries doing in-order scans of database tables as a sequential workload. We populated a database at the scale factor of 1 (other scale factors were also tested but the results were quite similar). We ran these two workloads against this database, either in isolation or concurrently. We compare the total execution time for each workload, averaged across three runs. The results are presented in Figure 3. The random workload is labeled 'R', while the sequential workload is labeled 'S.' 'C1' bars show the execution time when each workload runs in isolation. while 'C2' bars show the time taken by each workload when the sequential and random benchmarks are run concurrently on the same disk.

Running the two workloads concurrently on the same disk increases the runtime of the random workload by only 10%: when run by itself, it takes 212 seconds, and when co-located with the sequential workload, it takes 233 seconds. However, co-location increases the runtime of the sequential workload more substantially, from 19 seconds to 51, an increase of 168%. This clearly demonstrates that random workloads do interfere with sequential workloads for real-world applications.

[3]TPC-H query 18 was removed because it took too long to finish.

3 System Design

The FAST system offers block-level storage abstraction to VMs. Each device is presented as a virtual disk, or a virtual volume, to the VM. The OS in the VM expects the volume have the same semantics of a raw disk, and have similar performance characteristics. In particular, sequential IOs in the linear address space of a volume should have much higher throughput than random IOs.

3.1 Assumptions

In designing a block storage system with predictable performance, we make the following assumptions.

- The system is built from many inexpensive commodity components that often fail. Replication is necessary to offer high availability and durability.
- Virtual volume sizes may range from 1GB to 1TB. The number of physical disks available in the cloud is much less than the total number of requested block devices from all tenants.
- A block device can only be attached to at most one VM at a given time as the exclusive reader and writer. Live VM migration between hosts is rare so that block device handoff does not need to be optimized. These assumptions make our system design different from many existing storage systems that have to make difficult tradeoff in design between consistency and performance.
- No assumption is made on the storage workload from VMs. The workload may be database queries made of many small random reads, or map-reduce jobs with mostly sequential IOs and some random IOs.
- All nodes in the cloud are within a single data center, interconnected by high speed Ethernet with enough bi-section bandwidth.

3.2 Architecture

There are three types of roles in our system architecture as shown Figure 4: compute nodes, name nodes, and data nodes. Each of these runs a commodity Linux machine.

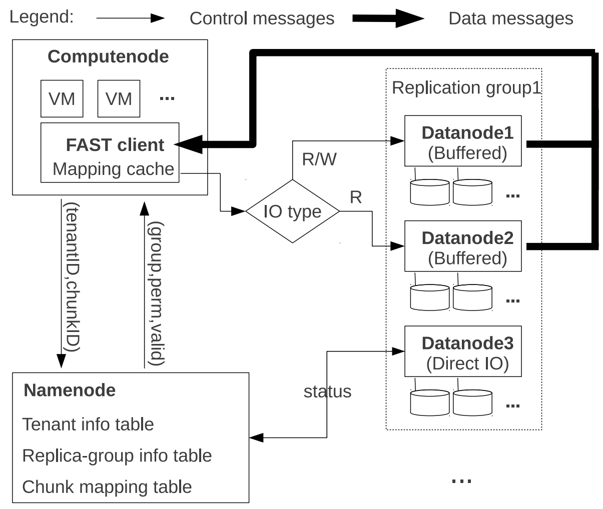


Figure 4: Overall System Architecture

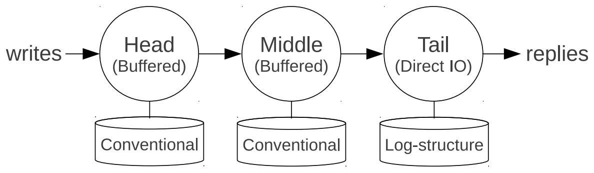


Figure 5: Chain Replication

A *compute node* runs a Xen hypervisor [4] that hosts VMs, and runs our block device driver in Domain-0 as the client of the storage service (the FAST client). All disk IO requests within the VM in Domain-U is redirected to the driver.

A *data node* stores the data of the virtual volumes. We partition every three data nodes into a replication group, in which data is replicated on every data node for reliability and availability. Volumes are divided into fixed-size chunks. Each chunk is stored in one replication group. A chunk is also lazily allocated when the first write request within the chunk range is issued. Neither the driver nor the data node caches volume data. Caches offer little benefit because the application and OS inside the VM already maintains them. The FAST clients do cache metadata, however.

Finally, a *name node* provides a metadata service. It manages the mappings from tenants and chunk IDs to the corresponding replication groups, replication group memberships, access control information, and leases from clients. It is also responsible for detecting failed data nodes via heartbeat messages. The name node is also in charge of allocating chunks and deciding where to place them. We choose to have a single name node to favor simple design with the same argument made in GFS [6], and leave more sophisticated architecture (e.g. Paxos based replication [13]) as future work.

When a volume is attached to a compute node, the driver first contacts the name node to establish a lease on the volume, then prefetches the chunk IDs of the volume and the addresses of the related replication groups. All these information is kept in memory. The lease will be periodically refreshed until the volume is detached. After initial setup, the driver starts to serve IO requests from the VM. If a write request is received, and the chunk has not been allocated, the client will ask the name node to allocate the chunk in a replication group. Otherwise the chunk location is available from the local lookup table. Once the chunk location is determined, the data is sent to the replica group in a chained fashion shown in Figure 5. If a read request is received, and the chunk has not been allocated, a zero-filled buffer is returned directly. Otherwise, we find the replication group of the chunk, and use our read algorithm in Section 3.3 to pick one data node out of three to retrieve the data, and then return to the VM. Any node in the replication group is able to handle it without inconsistency. This is different from the chain replication used in object storage [20] where multiple readers and writers are present. As the exclusive reader and writer, the device driver keeps track of pending write byte ranges and responds to the read requests only after pending overlapping writes finish.

3.3 Layout and Strategy

Each data node may use either conventional disk layout such that a chunk is stored in contiguous physical disk sectors, or a log-structured disk layout (log layout in short), in the same spirit of a log-structured file system [17]. In the log layout, data from different tenants are interleaved in the order of the time the data is written. A B-tree index is used to maintain a table to lookup data during read given a tuple specified by chunk ID, offset in chunk and length. Compared to the conventional layout (i.e., buffered write), the log layout effectively converts all chunk write requests, including both random and sequential, into sequential operations due to the append-only log structure. As a result, random writes from one tenant will not disruptively interfere sequential writes of another tenant who share the same physical disk.

For the three data nodes in a replication group, we designate the tail node of the replication chain to use the log layout, and the head and middle node to use the conventional layout. During replication, we only write data to the disk buffer cache on the first two nodes, but make sure to flush all data to the disk on the tail node before a write succeed reply is sent to the compute node. The write strategy guarantees that the data is replicated on all three nodes, and is persistent on the tail node at least. The probability of all data corruption on three nodes at the same time due to power loss, server crash, or malfunctioning disk hardware is very low.

We choose *not* to flush buffer cache on the head and middle data node immediately after write requests for two reasons. First, write requests from one tenant may collide with read requests from another tenant. Having the option to delay writes without sacrificing much durability due to replication gives us the opportunity to address the read-write conflict. Second, because we use the conventional layout on those two nodes, flushing each write request would cause write-write interference among tenants.

For each read request, we use a *default-with-steal* strategy to select the data node for read. By default, the device driver always sends random read requests to the head node, and sequential read request to the middle node. As such, the read workload is always competing with the same type of workload, thus receiving its fair share. However, we also allow a data node to steal read requests from others, if it is currently idling or very lightly loaded compared to others, to maximize the disk utilization. For example, if the tail node is not doing any writes, or the middle node is not doing any sequential reads, they may steal random read requests from the head node queue to improve performance. Note that the device driver itself cannot make the decision because it is only aware of the workload of its own tenant.

4 Initial Results

We use simulation to demonstrate the advantage of our design. We compare FAST with a traditional approach called Baseline. In Baseline, chunks are also replicated in the replication group three times, but every data node uses the conventional disk layout. The IO scheduler is not workload type aware. Instead it binds a tenant to a replica for read in a round-robin fashion for load balancing.

In the simulation, we study one replication group with 3 disks and 30 tenants. Each has exactly one chunk assigned in the group, and all start workloads at the same time. The workloads consist of 10 random read of 16 MB each (RR), 10 sequential reads of 19 MB each (SR) , 5 random writes of 20 MB (RW), and 5 sequential writes of 20 MB (SW). For Baseline, 4 RRs and 3 SRs are assigned to the head node, 3 RRs and 4 SRs to the middle node, and 3 RRs and 3 SRs to the tail node. In FAST, 10 RRs are assigned to the head node, and 10 SRs to the middle node. All SWs and RWs are performed on the head and middle node as buffered writes, and on the tail node as direct write. However, note that in FAST the 5 RWs and the 5 SWs will be converted to one single SW workload on the tail node (the one with the log structure).

We used the same disks and tools described in Section 2 to simulate the behavior of FAST and Baseline. We measured the elapsed time for each workload and present the results in Figure 6. Each bar represents the performance for one workload. The bars marked as D1, D2 and D3 are workload performance in Baseline sched-

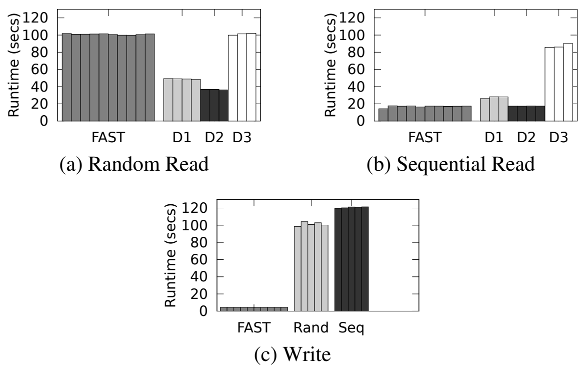


Figure 6: FAST vs. Baseline for different workloads

uled on the head, middle and tail node respectively. We observe that (1) for the same type of workload, FAST gives consistent and fair performance regardless of the tenancy. Baseline yields highly variable performance depending on which node the tenant is associated with; (2) for sequential read and write workloads, FAST outperforms Baseline because there is no disk seek caused by random workloads for disruption; (3) for random write requests, because FAST adopts the log layout, it significantly outputs Baseline; (4) random read workloads take longer to execute in FAST than in Baseline. This is because those workloads are fixed on the head node in our simulation for FAST. The middle node and the tail node have been idle since 20 seconds and 4 seconds respectively. We expect the performance of random read in FAST to improve greatly once we implement the steal strategy (Section 3.3) for load balancing so that the two nodes can share the workload of the head node once they are idle. But more importantly, FAST achieves a fair share of system resource with strong and consistent predictability while Baseline fails to do so.

5 Related Work

There are many related work on providing QoS-based resource allocation for storage, such as Stonehenge [11], Argon [21], and Aqua [12]. The goal of these algorithms is to allocate throughput or bandwidth in proportion to the pre-specified weights of the clients. Further proposals provide additional support for latency-sensitive applications (SMART [15], BVT [5], pClock [8]). Furthermore, mClock [9] borrows the concept of reservation and limit from CPU and memory scheduling to storage in additional to the proportional weight based allocation. These work typically abstract the storage device to a single block device, such as a physical disk, a LUN or a RAID device. They rely on the lower layer to deal with replications and do not try to change the I/O behavior of the guest VMs or clients. In contrast, we propose to leverage the replication information during scheduling to optimize the performance while maintaining fairness, and use log-structured storage on selected replicas to avoid write workload conflict.

6 Conclusion and Ongoing Work

In this position paper, we reveal the performance interference problem among different tenants when co-locating different types of workloads in the cloud. Then we propose the design of a block storage system FAST which provides a much better fair-sharing and a higher or at least comparable performance for different workloads. Our initial results showed that the new design is effective and promising.

We are still at the early stage of the design and implementation of FAST. There are many interesting challenges we have not investigated. First, in our initial evaluations, we assume that all workloads belonging to one type have similar I/O request characteristics. We plan to investigate effects of co-locating same type of workloads but with different I/O request characteristics (e.g., in term of block size and request frequency) and build models for them. Second, we need to deal with failures, including data and name node crashes and disk corruption. Because of different disk layouts and roles, the recovery procedures for data nodes might be different. Third, for load-balancing purpose, we need to design an allocation and placement algorithm on name node to efficiently use physical resources. We might also need to consider live chunk migration across replication groups to reduce hot spot. Forth, we need to study the tradeoff of chunk size selection with realistic workload. If the size is too small, then sequential workloads will be affected. But if the size is too large, it is likely to cause hot spot in a replica group. Finally, we are in the process of building the system. We expect the interaction of the network layer and the virtualization layer with the storage layer will also impose research challenges such as performance anomalies.

References

[1] I. Ahmad, J. M. Anderson, A. M. Holler, and V. M. Rajit Kambo. An analysis of disk performance in vmware esx server virtual machines. In *Proc. WWC*, Oct. 2003.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, 2009.

[3] J. Axboe. Flexible io tester. `http://freecode.com/projects/fio`.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. SOSP*, pages 164–177, 2003.

[5] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proc. SOSP*, 1999.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. SOSP*, pages 29–43, Oct. 2003.

[7] A. Gulati, C. Kumar, and I. Ahma. Storage workload characterization and consolidation in virtualized environments. In *Proc. VPACT*, Apr. 2009.

[8] A. Gulati, A. Merchant, and P. J. Varman. pclock: an arrival curve based approach for qos guarantees in shared storage systems. In *Proc. SIGMETRICS*, 2007.

[9] A. Gulati, A. Merchant, and P. J. Varman. mclock: handling throughput variability for hypervisor io scheduling. In *Proc. OSDI*, 2010.

[10] J. G. Hansen and E. Jul. Lithium: Virtual machine storage for the cloud. In *Proc. SoCC*, pages 15–26, 2010.

[11] L. Huang. Stonehenge: a high-performance virtualized ip storage cluster with qos guarantees. Technical report, 2003.

[12] L. Krishnamurthy. Aqua: an adaptive quality of service architecture for distributed multimedia applications. Technical report, 1997.

[13] L. Lamport. Paxos made simple. In *ACM SIGACT News 32*, pages 18–25, Dec. 2003.

[14] S. J. Nadgowda and R. Sion. Cloud performance benchmark series: Amazon ebs, s3, and ec2 instance local storage. Technical report, Oct. 2010.

[15] J. Nieh and M. S. Lam. A smart scheduler for multimedia applications. *ACM Transactions on Computer Systems*, 21(2):117–163, May 2003.

[16] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. CCS*, pages 199–212, 2009.

[17] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *ACM TOCS*, pages 26–52, Feb. 1992.

[18] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. In *Proc. VLDB*, pages 460–471, Sept. 2010.

[19] Transaction Processing Performance Council. The tpc-h benchmark. `http://www.tpc.org/tpch/`, 2012.

[20] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. OSDI*, pages 91–104, Dec. 2004.

[21] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *Proc. FAST*, Feb. 2007.