Towards General-Purpose Resource Management in Shared Cloud Services

Jonathan Mace[†] *Peter Bodik*[‡] *Rodrigo Fonseca*[†] *Madanlal Musuvathi*[‡]
[†]Brown University [‡]Microsoft Research

Abstract

In distributed services shared by multiple tenants, managing resource allocation is an important pre-requisite to providing dependability and quality of service guarantees. Many systems deployed today experience contention, slowdown, and even system outages due to aggressive tenants and a lack of resource management. Improperly throttled background tasks, such as data replication, can overwhelm a system; conversely, high-priority background tasks, such as heartbeats, can be subject to resource starvation. In this paper, we outline five design principles necessary for effective and efficient resource management policies that could provide guaranteed performance, fairness, or isolation. We present Retro, a resource instrumentation framework that is guided by these principles. Retro instruments all system resources and exposes detailed, real-time statistics of per-tenant resource consumption, and could serve as a base for the implementation of such policies.

1 Introduction

Today, many distributed services are *shared* by multiple tenants, both on private and public clouds and datacenters. These include common storage, data analytics, database, queuing, or coordination services like Azure Storage [4], Amazon SQS [1], HDFS [26], or Hive [30]. While sharing these services across tenants has clear advantages in terms of cost, managing the underlying resources is challenging.

Ideally, multi-tenant service providers should be able to implement resource management policies with various high-level goals – *e.g.*, admission control, fairness, guaranteed performance, or usage limits. These policies enable the provider to guarantee service-level objectives (SLOs) to a client, while simultaneously supporting other clients with differing workload characteristics. Equally important, these policies can ensure that a client does not trigger a system-wide outage by adversarially or inadvertently starving essential background tasks of required resources.

Traditionally, resource isolation has been enforced using OS-level primitives at the granularity of processes or users (*e.g.*, cgroups [5]) or using hypervisors that provide similar isolation among virtual machines. There is also some progress in providing network performance guarantees to groups of VMs [24, 20].

However, in distributed systems there is a mismatch in granularity between resource management and the existing mechanisms: on the one hand tenants share the same processes, thus using the same data structures, thread pools, and locks; on the other hand, several processes spanning machines work on behalf of the same tenant, requiring coordinated management. Prior approaches rely on ad-hoc enforcement techniques that are difficult to apply to other systems [33, 25].

In this paper, we consider some of the challenges faced by resource management policies, which we observed in practice: 1) due to extensive APIs, a system can bottleneck on any hardware (*e.g.*, disk) or software resource (*e.g.*, locks); 2) because users share resources at the application level, it may not be apparent which user is responsible for system load; 3) tenants interfere not only among themselves but with potentially expensive internal system tasks; 4) the resource requirements for each request issued by a client can vary substantially based on its type, arguments, and the system state; and 5) it can be unpredictable on which machines a request will execute and for how long.

While ignoring some of these challenges can lead to resource management policies that work in restricted scenarios (§5), we argue that one has to consider all of them to build resource management policies that are *effective*, *i.e.*, that work in a general setting, and *efficient*, *i.e.*, that achieve their objectives without being overly aggressive and wasting resources. These observations motivate five design principles necessary to implement such policies (§2), and the design of Retro (§3), our prototype framework for resource tracking and enforcement in distributed systems. Retro tracks the tenants of a system across a comprehensive set of resources, exposes usage statistics in realtime, and provides hooks back into the system to effect resource management decisions. Retro's design is guided by the principles we outline, and our preliminary evaluation (§4) shows evidence that it could be used by policies to properly manage resources in a shared distributed system.

2 Resource Management Design Principles

This section describes design principles that are necessary for a resource management policy to be effective and efficient. The principles are motivated by our experiences and observations from multiple data, compute, and communication oriented systems. For concreteness we present our observations in the context of HDFS.

Observation: Multiple request types can contend on unexpected resources. For our purposes, it suffices to say that an HDFS cluster has a single NameNode (NN) that manages the metadata for the filesystem, and many DataNodes (DNs) that store replicated file blocks.

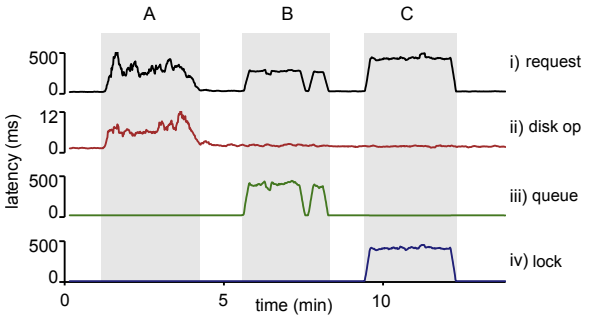

Figure 1: i) Request latency for a tenant reading 8KB files from HDFS [26] with intermittent background workloads A) replicating HDFS data, B) listing large directories, and C) making new directories; ii) latency of DataNode disk operations, iii) latency at NameNode RPC Queue, iv) latency to acquire NameNode "NameSystem" lock.

Since the core functionality of HDFS involves reading and writing files from DNs, one could consider disk and network as the *primary resources* that require explicit resource management. On the other hand, HDFS contains many other request types – in fact, there are over 100 API calls [15] – such as listing files in a directory, creating directories, creating symbolic links, or renaming files.

Figure 1 demonstrates how the latency of an HDFS client can be adversely affected by other clients executing very different types of requests, contending at different resources such as queues and locks. That experiment was run in a very simple setup with one NN and one DN, running on two machines. However, the ability of a single tenant to adversely affect other tenants' performance generalizes beyond this simple scenario. For instance, a Hadoop job that reads many small files can stress the storage system with disk seeks, like workload A in the figure, and impact all other tenants using the disks. Similarly, a tenant that repeatedly resubmits a job that fails quickly puts a large load on the NN, like workload B, as it has to list files in the job input directories. In communication with Cloudera [32], they acknowledge several instances of aggressive tenants impacting the whole cluster, saying "anything you can imagine has probably been done by a user".

The bottleneck resource in each of these instances varies from locks, thread pool queues, to the storage and the network. While it might be tempting to design throttling and scheduling policies based only on the primary APIs and resources, our experiments show that this would be incomplete. Thus, robust resource management *requires* a comprehensive accounting of all resources that clients can potentially bottleneck on, and consideration of all possible API calls. Our first principle comes from this.

Principle: Consider all request types and all resources in the system

Observation: Contention may be caused by only a subset of tenants. Distributed systems comprise multiple processes across many machines, and different tenants contribute different load to the system. Resource contention may be localized to a subset of machines or resources. Some tenants may not be accessing these machines or resources, while other tenants may be consuming more than their fair share. If a goal of the system was to reduce contention on these resources, it would be inefficient and unfair to penalize all tenants equally when only a subset may be culpable.

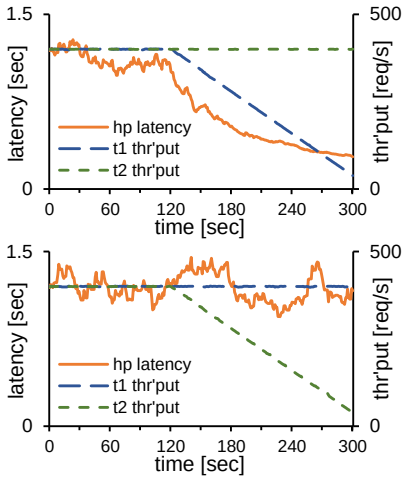

Figure 2: The figures show the impact of manually throttling two background tenants (t1,t2), demonstrating that only t1 impacts the high priority tenant (hp)

Figure 2 demonstrates the effect in HDFS on the latency of a high priority tenant, when we manually throttle the request rates of two other tenants. The figure shows a high-priority tenant, t_{hp}, sending 4MB write requests, sharing the service with two low-priority tenants. Tenant t_1 submits 8kB random reads, while tenant t_2 lists files in a directory. When we separately throttle the request rates of the background tenants, we observe an effect on the latency of t_{hp} only when throttling t_1.

In the above example, if our goal was to decrease the latency of t_{hp}, we would only benefit from reducing t_1's request. A non-trivial system should be capable of *targeting* the cause of contention - the tenants, machines, and resources responsible. This motivates our second principle.

Principle: Distinguish between tenants.

Observation: Foreground requests are only part of the story. Many distributed systems perform background tasks that are not directly triggered by tenant requests, but compete for the same resources. For instance, HDFS performs data replication after failures, asynchronous garbage collection after file deletion, periodic DN heartbeats, and more. These background tasks can adversely affect the performance of foreground tasks. Jira HDFS-4183 [14] describes an example of NN overload when a large number of files are abandoned without closing, which triggers a storm

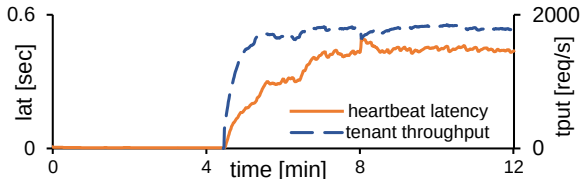

Figure 3: Heartbeat latency is initially 5ms. Heartbeat latency jumps to 500ms when the tenant's workload is introduced.

of block recovery operations after the lease expiration interval one hour later. Guo et al. [13] describe a failure in Microsoft's datacenter where a background task spawned a large number of threads, overloading the servers.

On the other hand, critical system-generated tasks need to be protected *from* foreground tasks. Guo et al. [13] describe a cascading failure resulting from overloaded servers not responding to heartbeats, triggering further data replication and overload. Figure 3 shows how HDFS DN heartbeat latency increases from 5ms to nearly 500ms when a tenant overloads the shared NN thread pool.

From these observations we derive our second principle. Resource management policies should treat both system- and client-generated tasks as first class entities.

> *Principle: Treat foreground and background tasks uniformly.*

Observation: Resource demands are very hard to predict. Many schedulers [12, 28, 10] need the *cost* of a request to be specified a priori, often in a multidimensional space representing the different resources.

We argue that resource requirements estimated offline would be insufficient for a number of reasons; a) the resources requested by a task could be influenced by one or more of the arguments of the API call; b) a model would need to encode both the *total* cost and the *rate* of resource consumption; c) the presence of other tenants could influence the behavior of a request (eg, by evicting cache entries); d) in order to handle localized congestion a model would need to know which machines will execute a request; and e) the state of the system can affect the success of operations (eg, renaming a non-existent file).

> *Principle: Estimate resource usage at runtime.*

Observation: Requests can be long or lose importance. Admission control is a common example of resource management, whereby requests are admitted or rejected at entry to the system based on their perceived impact. However, in systems with partitioned data, a large set of requests might be directed to the same disk holding a popular piece of data, creating a hot spot. While at the entry point, the overall load of the requests might seem small in comparison to the system's total capacity, the localized load introduced by the requests could be substantial if all directed to a single machine (*e.g.*, if all read from a single DN).

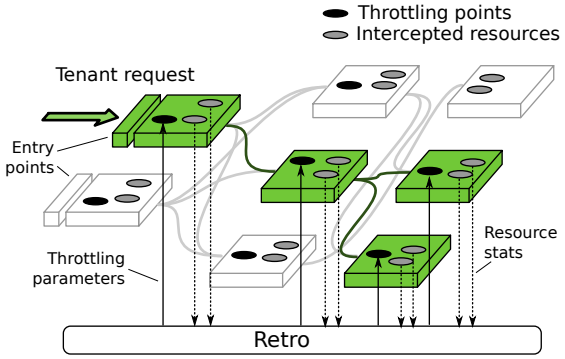

Figure 4: Retro architecture with the execution path of a tenant's request highlighted. Metadata is propagated alongside requests as they traverse the system. Resources are intercepted during execution and statistics reported via Retro. Retro exposes throttling points for developers to impose schedulers on the system.

Additionally, the duration of one API call or background task can vary substantially; an HDFS rename call executes in a matter of milliseconds, but writing a 128MB data block takes many seconds even in an uncontended system. Once admitted, a long request will continue to consume resources until completion. If a tenant of higher priority were to suddenly start competing for resources, the lower priority requests already admitted to the system would contend with the high priority tenant for a potentially long period of time. In this case, the system should be able to intervene to throttle, pause or cancel the lower priority requests if necessary.

Our last principle stems from this: a resource management policy should be able to act on requests while they are in-flight.

> *Principle: Schedule early, schedule often.*

3 Retro Instrumentation Platform

Based on the challenges and principles outlined in the previous section, we argue that an efficient resource management system will require detailed and timely tracking of the resources used by each tenant and each request. This motivates the design of Retro, a prototype framework for resource tracking and throttling in distributed systems. At a high level, Retro collects per-tenant, per-request resource usage at different points as the system executes, both within processes and across nodes. It then aggregates these in near real time. Tenants in Retro can be both foreground clients and background tasks. Finally, Retro provides hooks to throttle particular tenants or requests. Figure 4 shows Retro's architecture.

While Retro does not currently perform any action automatically, it is the first step towards a complete system that implements resource management policies that are robust and efficient. We describe Retro in more detail below, and some early results in §4 suggest that Retro can inform and help enforce such policies with acceptably low overhead.

Tenant abstraction Retro uniformly abstracts foreground clients and background tasks as *tenants*. A tenant is a collection of tasks that serve some common purpose, such as tasks from a specific user account, heartbeat requests to detect failed nodes, or data replication tasks. In Retro, a tenant is the granularity of resource attribution and management, and is identified by a unique ID.

End-to-End ID Propagation Borrowing from end-to-end causal tracing [8, 22, 27, 29], Retro propagates tenant IDs along the execution path of a request, such that at any point in time, the instrumentation knows on behalf of which tenant an operation is executing. At the beginning of a request, Retro associates the thread executing the request with a tenant by storing the tenant ID in a thread local variable, removing this association when the request completes. The resource instrumentation queries this variable to charge the resource usage to a specific tenant.

Automatic Resource Instrumentation using AspectJ Retro uses AspectJ [17] to automatically instrument all hardware resources and resources exposed through the Java standard library. It captures disk and network usage by intercepting constructor and method calls on file and network streams; it tracks CPU usage by the time a thread is associated with a tenant using `QueryThreadCycleTime` in Windows or `clock_gettime` in Linux. To capture locking we instrumented Java monitor locks and all implementers of the `Lock` interface, while we instrumented thread pools using Java's `Executors` framework. The only manual instrumentation required is for application-level resources created by the developer, such as custom queues, thread pools, or pipeline processing stages.

Aggregation and Reporting When a resource is intercepted, Retro determines the tenant associated with the current thread, and increments in-memory counters that track the per-tenant resource use. These counters include the number of resource operations started and ended, total latency spent executing in the resource and any operation-specific statistics such as bytes read or queue time. A separate thread reads the counters and reports the values to any subscribed clients at a regular interval.

Entry and Throttling Points Retro exposes hooks for a developer to specify entry and throttling points in their application. An entry point establishes the tenant initiating the execution. For example, in our HDFS instrumentation we added entry points in the HDFS client API, and in code that initiates background tasks. A throttling point can impose a scheduler on queues in the system, rate-limit a tenant by pausing threads, or take advantage of more sophisticated mechanisms such as distributed rate limiters [21]. For example, in our HDFS instrumentation we added throttling points at the high-level system entry points, the NameNode RPC queue, and in the DataNode block transfer threads.

operation	latency
Deserialize metadata	80ns
Read active metadata	9ns
Serialize metadata	46ns
Record use one resource operation	342ns

Table 1: Costs of Retro instrumentation

Retro satisfies each of the design principles outlined in §2: by measuring resource consumption at runtime, it exposes the resources actually being consumed by a tenant, eschewing the need for a priori models of request types. Retro tracks multiple resources (it is extensible to new types of resource), and end-to-end tracing allows it to distinguish the tenant responsible for resource usage within and across processes. The entry point mechanism allows Retro to treat foreground and background tasks uniformly, and throttling points allow it to impose scheduling decisions in multiple places on the execution path.

4 Evaluation on HDFS

We instrumented HDFS with Retro prototype, and show early evidence that it can be useful for resource management policies, while keeping low overhead. We also show integrating Retro presents a low burden for developers.

Experiments The experiments outlined in §2 give examples of how Retro could be useful to resource management policies, showing how it exposes granular information to identify bottlenecks, how the cause of a bottleneck can then be targeted, and how both foreground and background tasks can be acted upon. Figure 1 demonstrates per-tenant and per-resource statistics that Retro can record in real-time. A policy could easily identify the bottleneck resource for each of the three different background workloads, and which tenants are contending on that resource. Figure 2 shows two different manual interventions using Retro's throttling points, and demonstrates how an efficient resource management policy could target only the tenant responsible for congestion in order to achieve some high level goal. Figure 3 demonstrates that policies could act on client requests, or on internal system tasks, taking advantage of how Retro treats both as first-class entities.

Overhead of Retro. Retro propagates a tenant ID (3 bytes) along the execution path of a request, incurring up to 80ns of overhead (see Table 1) to serialize and deserialize when making network calls. The overhead to record a single resource operation is approximately 340ns, which includes intercepting the thread, recording timing, CPU cycle count (before and after the operation), and operation latency, and aggregating these into a per-tenant report. To estimate the impact of Retro on throughput and end-to-end latency, we benchmark HDFS and HDFS instrumented with Retro using three different request types (rename, and reads of 8kB and 4MB), see Figure 5 for results across four 20-

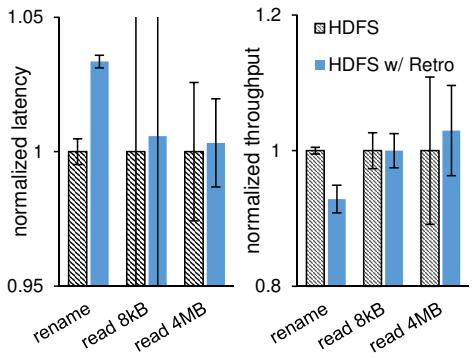

Figure 5: Normalized latency (left) and throughput (right) for rename and read operations comparing HDFS to HDFS instrumented with Retro, along with error bars showing one standard deviation across four runs. Standard deviation in both read 8kB experiments is about 0.1.

minute runs. Per-request latency increases the most for the rename API, by 3.5%, because it performs in-memory operations on the NameNode with a comparatively high number of trace points; its throughput drops by 7.7%. The read operations slow down by 0.6% and 0.3%, respectively, because they spend most of their time reading from disk. The average throughput of read 4MB increases with Retro, most likely due to the large variance observed.

Developer Effort While Retro requires manual developer intervention to propagate tenant IDs across network boundaries and to verify correct behavior of Retro's automatic instrumentation, our experience shows that this requires little work. Instrumenting resource operations for HDFS was handled automatically using AspectJ. To instrument HDFS we only added about 150 lines of code. We manually instrumented HDFS's Protocol Buffer [11] messages, and data transfer packets, to include Retro metadata. We specified entry points at the client RPC server on the NN, and in the source code where heartbeats, block replication, and block invalidation was initiated. We also placed throttling points at these entry points.

5 Related Work

Many shared distributed systems implement some variant of performance isolation, such as fair sharing [25, 28, 9], throttling aggressive tenants [4, 26], and providing latency or throughput guarantees [33, 34, 28, 7]. Some systems drop or queue requests at client-facing entry points or at machine boundaries, though with great variation in the granularity of decisions. High-level schedulers such as Mesos [16], Yarn [31], or Sparrow [19] centrally allocate tasks to machines. Some cloud storage systems, *e.g.*, S3 [23] or Azure Storage [4], make admission decisions for each individual API call. Others have proposed distributed rate limiters, at the network [28, 21] and disk [12] layers, to enforce global reservations, limits and shares on IO.

Lack of visibility of actual resource bottlenecks leads to the ad-hoc selection of the metrics used for performance isolation. For example, Azure Storage [4] and Pisces [25] select only request rate and operation size as metrics. SQLVM [18] uses CPU, I/O, and memory as key resources. Cake [33] breaks HDFS requests into equal-sized chunks, then assumes disk as a bottleneck and uniform cost for each chunk. In our communication with operators running planet-scale cloud services at Microsoft, unexpectedly long-running requests in new workloads often force adjustments to admission control rules and mechanisms.

In some cases, careful system design or restricted operating environments can obviate some of our observations. For example, some approaches benefit from having fixed- or bounded-sized requests; IOFlow [28] provides guarantees for networked storage, and enforces them at the network packet level; Cake [33] explicitly sub-divides large HDFS reads into smaller, equal-sized chunks. Amazon's DynamoDB [7] hedges that uniform load distribution and flexible durability guarantees are sufficient to satisfy client latency requirements.

In all cases we observed, the enforcement mechanisms for high-level policies were manually implemented. For example, Cake [33] manually instruments the RPC entry points of HDFS and HBase to add queues and associates tenants based on an identifier from the HDFS RPC headers; IOFlow [28] modifies queues in key resources (*e.g.*, NIC, disk driver) on the data path; and Pisces [25] modifies the scheduling and queueing code of Membase and directly updates tenant weights at these queues.

Banga and Druschel addressed the mismatch between OS abstractions and the needs of resource accounting with resource containers [2], which, albeit in a single machine, aggregate resource usage orthogonally to processes, threads, or users. Retro achieves per-tenant, distributed resource accounting by combining previous results in resource monitoring [6], automatic source code instrumentation [17], and end-to-end metadata propagation [3, 8, 27]. Retro extends the notion of a resource to arbitrarily include hardware and software resources and its throttling point abstraction can automatically insert mechanisms such as distributed rate limiters at resources.

6 Conclusion

In this work we presented Retro, a framework for effecting resource management decisions in distributed systems. Retro tackles important challenges in this direction: it addresses the requirements of a robust and efficient resource management mechanism in shared distributed systems, and is a practical approach with low overheads. We view Retro as a step to developing broader resource management policies that are applicable to a variety of systems. It offers a platform for future work to develop such policies, and guidance to designers of alternative mechanisms.

References

[1] Amazon web services. http://aws.amazon.com/. Accessed July 2014.

[2] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server systems. In *OSDI '99*, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modeling. In *Proc. USENIX OSDI*, 2004.

[4] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *SOSP '11*.

[5] Control Groups. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt. Accessed July 2014.

[6] G. Czajkowski and T. Von Eicken. Jres: A resource accounting interface for java. *ACM SIGPLAN '98*.

[7] Amazon DynamoDB. http://aws.amazon.com/dynamodb. Accessed July 2014.

[8] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. NSDI '07.

[9] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 1–12, New York, NY, USA, 2012. ACM.

[10] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *USENIX NSDI '11*.

[11] Google Protocol Buffers. http://code.google.com/p/protobuf/. Accessed July 2014.

[12] A. Gulati, A. Merchant, and P. J. Varman. mclock: Handling throughput variability for hypervisor io scheduling. In *OSDI '10*.

[13] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, Berkeley, CA, 2013. USENIX.

[14] HDFS-4183. http://bit.ly/1l4uWbu. Accessed July 2014.

[15] HDFS API. http://bit.ly/1cxFTD9. Accessed July 2014.

[16] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. NSDI'11.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[18] V. R. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri. Sqlvm: Performance isolation in multi-tenant relational database-as-a-service. In *CIDR'13*. www.cidrdb.org, 2013.

[19] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[20] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: Sharing the network in cloud computing. In *SIGCOMM*, 2012.

[21] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud control with distributed rate limiting. In *ACM SIGCOMM*.

[22] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: detecting the unexpected in distributed systems. In *NSDI'06*, Berkeley, CA, USA, 2006. USENIX Association.

[23] Amazon Simple Storage Service. http://aws.amazon.com/s3. Accessed September 2014.

[24] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: Performance isolation for cloud datacenter networks. HotCloud '10.

[25] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI '12*.

[26] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[27] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[28] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. Ioflow: A software-defined storage architecture. SOSP '13.

[29] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking activity in a distributed storage system. *SIGMETRICS '06*.

[30] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE '10*.

[31] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. Proc. SoCC '13. ACM.

[32] A. Wang. personal communication.

[33] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: Enabling High-level SLOs on Shared Storage Systems. In *SoCC '12*.

[34] A. Wang, S. Venkataraman, S. Alspaugh, I. Stoica, and R. Katz. Sweet Storage SLOs with Frosting. HotCloud '12.