Retro: Targeted Resource Management in Multi-tenant Distributed Systems

Jonathan Mace[1], Peter Bodik[2], Rodrigo Fonseca[1], Madanlal Musuvathi[2]
[1]Brown University, [2]Microsoft Research

Abstract

In distributed systems shared by multiple tenants, effective resource management is an important pre-requisite to providing quality of service guarantees. Many systems deployed today lack performance isolation and experience contention, slowdown, and even outages caused by aggressive workloads or by improperly throttled maintenance tasks such as data replication. In this work we present Retro, a resource management framework for shared distributed systems. Retro monitors per-tenant resource usage both within and across distributed systems, and exposes this information to centralized resource management policies through a high-level API. A policy can shape the resources consumed by a tenant using Retro's control points, which enforce sharing and rate-limiting decisions. We demonstrate Retro through three policies providing bottleneck resource fairness, dominant resource fairness, and latency guarantees to high-priority tenants, and evaluate the system across five distributed systems: HBase, Yarn, MapReduce, HDFS, and Zookeeper. Our evaluation shows that Retro has low overhead, and achieves the policies' goals, accurately detecting contended resources, throttling tenants responsible for slowdown and overload, and fairly distributing the remaining cluster capacity.

1 Introduction

Most distributed systems today 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 [6], Amazon SQS [3], HDFS [36], or Hive [41]. Multi-tenancy has clear advantages in terms of cost and elasticity.

However, providing performance guarantees and isolation in multi-tenant distributed systems is extremely hard. Tenants not only share fine-grained resources within a process (such as threadpools and locks) but also resources across multiple processes and machines (such as the disk and the network) along the execution path of their requests. As a result, traditional resource management mechanisms in the operating system and in the hypervisor are ineffective due to a mismatch in the management granularity. Moreover, tenant-generated requests not only compete with each other but also with system-generated tasks, such as replication and garbage collection, for shared resources. In addition, the bottleneck responsible for degrading the performance of a tenant can change in unpredictable ways depending on its input workload, the workload of other tenants and system tasks, the overall state of the system (including caches), and the (nonlinear) performance characteristics of underlying resources. See Figure 1 for an example. It does not help that the APIs to these services are often complex, with HDFS, for example, having over 100 calls in its client library [19], making static workload models intractable.

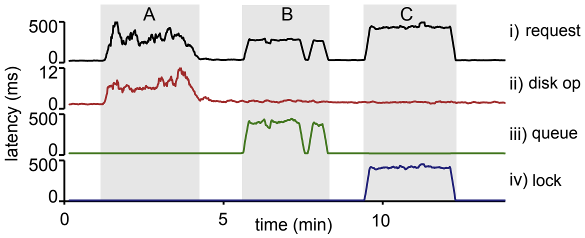

Figure 1: i) Latency for a client reading 8kB files from HDFS [36] is impacted by different workloads that A) replicate HDFS blocks, B) list large directories, and C) make new directories, each overloading the disk, threadpool, and locks respectively. ii) latency of DataNode disk operations, iii) latency at NameNode RPC queue, iv) latency to acquire NameNode "NameSystem" lock.

We address these challenges with Retro, a resource management framework whose core principle is to separate resource management policies from the mechanisms required to implement them. Retro enables system designers to state, verify, tune, and maintain management policies independent of the underlying system implementation. As in software defined networking, Retro policies execute in a *logically-centralized* controller with Retro mechanisms providing a global view of resource usage both within and across processes and machines.

The goal of Retro is to enable *targeted* policies that achieve desired performance guarantee or fairness goals by identifying and only throttling the tenants or system activities responsible for resource bottlenecks. Retro provides three abstractions to simplify the development of such policies. First, it groups all system activities – both tenant-generated requests and system-generated tasks – into individual *workflows*, which form the units of resource management. Retro attributes the usage of a resource at any instant to some workflow in the system. Second, Retro provides a *resource* abstraction that unifies arbitrary resources, such as physical storage, network, CPU, thread pools, and locks, enabling resource-agnostic policies. Each resource exposes two opaque performance metrics: *slowdown*, a measure of resource contention, and a per-workflow *load*, which attributes the resource usage to workflows. Finally, Retro creates *control points*, places in the system that implement resource scheduling mechanisms such as token buckets, fair schedulers, or priority queues. Each control point schedules requests locally, but is configured centrally by the policy.

Retro advocates *reactive* policies that dynamically respond to the current resource usage of workflows in the system, instead of relying on static models of future resource requirements. These policies continuously react to changes in resource bottlenecks and input workloads by making small adjustments directing the system towards a desired goal. Such a "hill climbing" approach enables policies that are robust to both changes in workload characteristics and nonlinear performance characteristics of underlying resources.

We evaluate Retro abstractions and design principles by implementing two fairness policies – reactive version of bottleneck resource fairness [12] and dominant resource fairness [13] – and LATENCYSLO, that enforces end-to-end latency targets for a subset of workflows. We use these policies on a Retro implementation for the Hadoop stack, comprising HDFS, Yarn, MapReduce, HBase and ZooKeeper. All three policies are concise (about 20 lines of code) and are agnostic of Hadoop internals. We experimentally demonstrate that these policies are robust and converge to desired performance goals for different types of workloads and bottlenecks.

The targeted and reactive policies of Retro rely on accurate, near real-time measurements of resource usage across all workflows and all resources in the system. Through a careful design of mostly-automatic instrumentation and aggregation of resource usage measurements our implementation of Retro for the Hadoop stack incurs latency and throughput overhead of 0.3% to 2%.

The goal of Retro is to be a general resource management framework that is applicable to arbitrary distributed systems. Our experience applying Retro to five distributed systems – HDFS, Yarn, MapReduce, HBase, and ZooKeeper – validates our design. Applying Retro to a new system required modest amounts of system-specific instrumentation – between 50 and 200 lines of code. The rest of the Retro framework required no changes. Moreover, resource management policies that we originally developed for HDFS were directly applicable to other systems, validating the robustness of Retro abstractions.

In summary, our key contributions are:

- Unifying abstractions of *workflows*, *resources*, and *control points*, that enable concise policies that are system-agnostic and resource-agnostic;
- Demonstrating the feasibility of Retro in a complex Hadoop stack, including a low-overhead, pervasive, per-workflow resource tracking and aggregation for a wide variety of resources;
- Targeted and reactive policies for providing latency SLOs, bottleneck resource fairness, and dominant resource fairness;
- A centralized controller that allow policies to enforce performance goals at different control points without requiring explicit coordination.

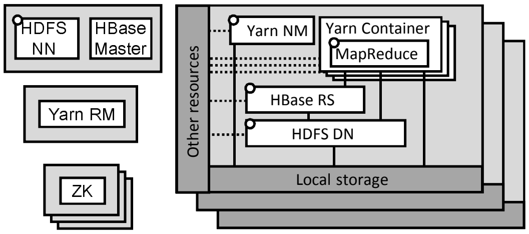

Figure 2: Typical deployment of HDFS, ZooKeeper, Yarn, MapReduce, and HBase in a cluster. Gray rectangles represent servers, white rectangles are processes, and white circles represent control points that we added. See text for details.

2 Motivation and challenges

This section motivates Retro by describing the challenges of resource management in a multi-tenant distributed system. As this paper presents Retro in the context of the Hadoop stack, we first provide a high-level overview of Hadoop components. The results of this paper generalize to other distributed systems as well.

2.1 Hadoop architecture

Figure 2 shows the relevant components of the Hadoop stack. HDFS [36], the distributed file system, consists of DataNodes (DN) that store replicated file blocks and run on each worker machine, and a NameNode (NN) that manages the filesystem metadata. Yarn [42] comprises a single ResourceManager (RM), which communicates with NodeManager (NM) processes on each worker. Hadoop MapReduce is an application of Yarn that runs its processes (application master and map and reduce tasks) inside Yarn containers managed by NMs. HBase [17] is a data store running on top of HDFS that consists of RegionServers (RS) on all workers and an HBase Master, potentially co-located with the NameNode or Yarn. Finally, ZooKeeper [21] is a system for distributed coordination used by HBase.

MapReduce job input and output files are loaded from HDFS or HBase, but during the job's *shuffle* phase, intermediate output is written to local disk by mappers (bypassing HDFS) and then read and transferred by NodeManagers to reducers. Reading and writing to HDFS has the

NameNode on the critical path to obtain block metadata. An HBase query executes on a particular RegionServer and reads/writes its data from one or many DataNodes.

2.2 Resource management challenges

Any resource can become a bottleneck Figure 1 demonstrates how the latency of an HDFS client can be adversely affected by other clients executing very different types of requests, contending for different resources. In production, a Hadoop job that reads many small files can stress the storage system with disk seeks, as workload A in the figure, and impact all other workloads using the disks. Similarly, a workload that repeatedly resubmits a job that fails quickly puts a large load on the NN, like workload C, as it has to list all the files in the job input directories. In communication with Cloudera [43], they acknowledge several instances of aggressive tenants impacting the whole cluster, saying "anything you can imagine has probably been done by a user". Interviews with service operators at Microsoft confirm this observation.

Multiple granularities of resource sharing On the one hand, concurrently executing workflows share software resources, such as threadpools and locks, within a process, while on the other hand, resources, such as the disk on Hadoop worker nodes, are distributed across the system. The disk resource, for example, is accessed by DN, NM, and mapper/reducer processes running across all workers. Systems have many entry points (*e.g.*, HBase, HDFS, or MapReduce API) and maintenance tasks are launched from inside the system. Finally, enforcing resource usage for long-running requests requires throttling inside the system, not just at the entry points.

Maintenance and failure recovery cause congestion Many distributed systems perform background tasks that are not directly triggered by tenant requests but compete for the same resources. E.g., HDFS performs data replication after failures, asynchronous garbage collection after file deletion, and block movement for balancing DN load. In some cases, these background tasks can adversely affect the performance of foreground tasks. Jira HDFS-4183 [18] describes an example where a large number of files are abandoned without closing, triggering a storm of block recovery operations after the lease expiration interval one hour later, which overloads the NN. Guo et al. [16] describe a failure in Microsoft's datacenter where a background task spawned a large number of threads, overloading the servers. On the other hand, some of these tasks need to be protected *from* foreground tasks. Guo et al. [16] describe a cascading failure resulting from overloaded servers not responding to heartbeats, triggering further data replication and further overload.

Resource management is nonexistent or noncomprehensive Systems like HDFS, ZooKeeper, and HBase do not contain any admission control policies. While Yarn allocates compute slots using a fair scheduler, it ignores network and disk, thus, an aggressive job can overload these resources. Interviews with service operators at Microsoft indicate that productions system often implement resource management policies that ignore important resources and use hardcoded thresholds. For example, a policy might assume that an `open()` is 2x more expensive than `delete()`, while the actual usage varies widely based on parameters and system state, resulting in very inaccurate resource accounting. The policies are often tweaked manually, typically *after* causing performance issues or outages, or when the system or the workloads change. Writing the policies often requires intimate knowledge of the system and of the request resource profile, which may be impossible to know a priori.

3 Design

The main goal of Retro is to enable simple, targeted, system-agnostic, and resource-agnostic resource-management polices for multi-tenant distributed systems. Examples of such policies are: a) throttle aggressive tenants who are getting an unfair share of bottlenecked resources, b) shape workflows to provide end-to-end latency or throughput guarantees, or c) adjust resource allocation to either speed up or slow down certain maintenance or failure recovery tasks.

Retro addresses the challenges in §2.2 by separating the mechanisms of measurement and enforcement of resource usage from high-level, global resource management policies. It does this by using three unifying abstractions – *workflows*, *resources*, and *control points* – that enable logically centralized policies to be succinctly expressed and apply to a broad class of resources and systems.

3.1 Retro abstractions

Workflow Resource contention in a distributed system can be caused by a wide range of system activities. Retro treats each such activity as a first-class entity called a *workflow*. A workflow is a set of requests that forms the unit of resource measurement, attribution, and enforcement in Retro. For instance, a workflow might represent requests from the same user, various background activities (such as heartbeats, garbage collection, or data load balancing operations), or failure recovery operations (such as data replication). The aggregation of requests into a workflow is up to the system designer. For instance, one system might treat all background activities as one workflow but another might treat heartbeats as a distinct workflow from other activities, if the system designer decides to provide a different priority to heartbeats.

Each workflow has a unique workflow ID. To properly attribute resource usage to individual workflows, Retro propagates the workflow ID along the execution path of all requests. This causal propagation [11, 37, 40, 34] allows

Retro to attribute the usage of a resource to a workflow at any point in the execution, whether within a shared process or across the network.

Resources A comprehensive resource management policy should be able to respond to a contention in any resource – hardware or software – and attribute load to workflows using it. A key hypothesis of Retro is that resource management policies can and should treat all resources, from thread pools to locks to disk, uniformly under a common abstraction. Such a uniform-treatment allows one to state policies that respond to disk contention, say, in the same way as lock contention. Equally importantly, this allows gradually expanding the scope of resource-management to new resources without policy change. For instance, a storage service might start by throttling clients based on their network or disk usage. However, as the complexity of the service increases to include sophisticated meta-data operations, the service can start throttling by CPU usage or lock-contention. On the other hand, the challenge in providing such a unifying abstraction is to capture the behavior of varied kinds of resources with different complex non-linear performance characteristics.

To overcome this challenge, Retro captures a resource's current *first-order* performance with two unitless metrics:

- **slowdown** indicates how slow the resource is currently, compared to its baseline performance with no contention;
- **load** is a per-workflow metric that determines who is responsible for the slowdown.

As a simple example, consider an abstract resource with an (unbounded) queue. Let $Q_{w,i}$ be the queueing time of the ith request from workflow w in a time interval and let $S_{w,i}$ be the time the resource takes to service that request. During this interval, the load by w is $\Sigma_i S_{w,i}$ and the slowdown is $\Sigma_{w,i}(Q_{w,i} + S_{w,i}) / \Sigma_{w,i} S_{w,i}$. Note, the denominator of the slowdown is the time taken to process the requests if the queue is empty throughout the interval.

The reactive policies in Retro allow these metrics to provide a linear approximation of the complex non-linear behavior. The policies continuously measure the resource metrics while making incremental resource allocation changes. Operating in such a feedback loop enables simple abstractions while reacting to nonlinearities in the underlying performance characteristics of the resource.

Resources in real systems are more complex than the simple queue above. Our goal is to hide the complexities of measuring the load and slowdown of different resources in *resource libraries* that are implemented once and reused across systems. See §4.2 for details.

An important implication of this abstraction is that it is not possible to query the capacity of a resource. Instead, a policy can treat a resource to have reached its capacity if the slowdown exceeds some fixed constant. Directly measuring true capacity is often not possible because of many request types supported (*e.g.* open, read, sync, etc. on a disk) and because of effects of caching or buffering, workflow demands do not compose linearly. Also, due to limping hardware [10], estimating the current operating capacity is next to impossible.

Control points To separate the low-level complexities of enforcing resource allocation throughout the distributed system, we introduce the *control point* abstraction. A control point is a point in the execution of a request where Retro can enforce the decisions of resource scheduling policies. Each control point executes locally, such as delaying requests of a workflow using a token bucket, but is configured centrally from a policy.

While a control point can be placed directly in front of a resource (such as a thread pool queue), it can more generally be located anywhere it is reasonable to sleep threads or delay requests, such as in HDFS threads sending and receiving data blocks. The location of control points should be selected by the system designer while keeping a few rules in mind. A control point should not be inserted where delaying a request can directly impact other workflows, such as when holding an exclusive lock. Conversely, some asynchronous design patterns (such as thread pools) present an opportunity to interpose control points, as it is unlikely that a request will hold critical resources yet potentially block for a long period of time.

Each logical control point has one or more instances. A point with a single instance is centralized, such as a point in front of the RPC queue in HDFS NameNode. Distributed points, such as in the DataNode or its clients, have many, potentially thousands of instances. Each instance measures the current, per-workflow throughput which is aggregated inside the controller.

To achieve fine-grained control, a request has to periodically pass through control points, otherwise, it could consume unbounded amount of resources. To illustrate this, consider a request in HBase that scans a large region, reading data from multiple store files in HDFS. If Retro only throttles the request at the RegionServer RPC queue, a policy has only one chance to stop the request; once it enters HBase, it can read an unbounded amount of data from HDFS and perform computationally expensive filters on the data server-side. By adding a point to the DataNode block sender, we can control the workflow at the granularity of 64kB HDFS data packets. More generally, the longer the period of time a request can execute without passing through a control point, the longer it will take any policy to react. This is similar to the dependence between the longest packet length L_{max} and the fairness guarantees provided by packet schedulers [31, 38].

3.2 Architecture

Figure 3 outlines the high-level architecture of Retro and its three main components. First, Retro has a measure-

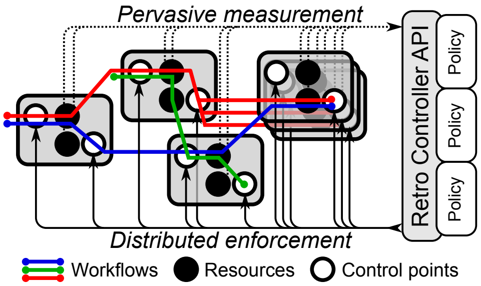

Figure 3: Retro architecture. Gray boxes are system components on the same or different machines. Workflows start at several points and reach multiple components. Intercepted resources (●) generate measurements that serve as inputs to policies. Policy decisions are enforced by control points (○).

ment infrastructure that provides near-real-time resource usage information across all system resources and components, segmented by workload. Second, the logically centralized controller uses the resource library to translate raw measurements to the load and slowdown metrics, and provides them as input to Retro policies. Third, Retro has a distributed, coordinated enforcement mechanism that consistently applies the decisions of the policies to control points in the system. We discuss the design of the controller in the following paragraphs. In §4 we describe the measurement and enforcement mechanisms in detail, and in §5 we present the implementation of three policies.

Logically centralized policies In current systems, resource management policies are hard-coded into the system implementation making it difficult to maintain as the system and policies evolve. A key design principle behind Retro is to separate the mechanisms (§4) from the policies (§5). Apart from making such policies easier to maintain, such a separation allows policies to be reused across different systems or extended with more resources.

Borrowing from the design of Software Defined Networks and IOFlow [39], Retro takes the separation a step further by logically centralizing its policies. This makes policies much easier to write and understand, as one does not have to worry about myopic local policies making conflicting decisions. In this light, we can view Retro as building a "control plane" for distributed systems, and providing a separation of concerns for policy writers and system developers and instrumenters.

Retro exposes to policies a simple API, shown in Table 1, that abstracts the complexity of individual resources and allows one to specify resource-agnostic scheduling policies, as demonstrated in §5. The first three functions in the table correspond to the three abstractions explained above. In addition, `latency(r,w)` returns the total time workflow `w` spent using resource `r`. `throughput(p,w)` measures the aggregate request rate of workflow `w` through a (potentially distributed) throttling point `p`, such as the entry point to the RS process. Finally, policies can affect the system through Retro's throttling mechanisms.

workflows()	list of workflows
resources()	list of resources
points()	list of throttling points
load(r,w)	load on `r` by workflow `w`
slowdown(r)	slowdown of resource `r`
latency(r,w)	total latency spent by `w` on `r`
throughput(p,w)	throughput of workflow `w` at point `p`
get_rate(p,w)	get the throttling rate of workflow `w` at point `p`
set_rate(p,w,v)	throttle workflow `w` at point `p` to `v`

Table 1: Retro API used by the scheduling policies. We omit auxiliary calls to set, for example, the reporting interval and smoothing parameters, as well as to obtain more details such as operation counts, etc.

4 Implementation

4.1 Per-workflow resource measurement

End-to-end ID propagation At the beginning of a request, Retro associates threads executing the request with the workflow by storing its ID in a thread local variable; when execution completes, Retro removes this association. While the developer has to manually propagate the workflow ID across RPCs or in batch operations, we use AspectJ to automatically propagate the workflow ID when using `Runnable`, `Callable`, `Thread`, or a `Queue`.

Aggregation and reporting When a resource is intercepted, Retro determines the workflow associated with the current thread, and increments in-memory counters that track the per-workflow 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. When the workflow ID is not available, such as when parsing an RPC message from the network, the resource use is attributed to the *next* ID that is set on the current thread (*e.g.*, after extracting the workflow ID from the RPC message). Retro does not log or transmit individual trace events like X-Trace or Dapper, but only aggregates counters in memory. A separate thread reads and reports the values of the counters to the central controller at a fixed interval, currently once per second. Reports are serialized using protocol buffers [14] and sent using ZeroMQ [2] pub-sub. The centralized controller aggregates reports by workflow ID and resource, smoothes out the values using exponential running average, and uses the resource library to compute resource load and slowdown.

Batching In some circumstances, a system might batch the requests of multiple workflows into a single request. HDFS NameNode, HBase RegionServers, and ZooKeeper each have a shared transaction log on the critical path of write requests. In these cases, we create a *batch workflow ID* to aggregate resource consumption of the batch task (e.g., the resources consumed when writing HBase transaction logs to HDFS). Constituent workflows report their relative contributions to the batch (e.g., serialized size of transaction) and the controller decomposes the resources consumed by the batch to the contributing workflows.

Automatic resource instrumentation using AspectJ Retro uses AspectJ [25] to automatically instrument all hardware resources and resources exposed through the Java standard library. Disk and network consumption is captured by intercepting constructor and method calls on file and network streams. CPU consumption is tracked during the time a thread is associated with a workflow. Locking is instrumented for all Java monitor locks and all implementers of the `Lock` interface, while thread pools are instrumented 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.

AspectJ is highly optimized and *weaves* the instrumentation with the source code when necessary without additional overheads. In order to avoid potentially expensive runtime checks to resolve virtual function calls, Retro instrumentation only intercepts constructors to return proxy objects that have instrumentation in place.

4.2 Resource library

Retro presents a unified framework that incorporates individual models for each type of resource. Management policies only make incremental changes to request rates allocated to individual workflows; for example, if the CPU is overloaded, a policy might reduce total load on the CPU by 5%. Therefore, as long as we correctly detect contention on a resource, iteratively reducing load on that resource will reduce the contention. Our models, thus, capture only the first-order impact of load on resource slowdown.

CPU We query the per-thread CPU cycle counter when setting and unsetting the workflow ID on a thread (using `QueryThreadCycleTime` in Windows and `clock_gettime` in Linux) to count the total number of CPU cycles spent by each workflow. The load of a workflow is thus proportional to its usage of CPU cycles. To estimate the slowdown, we divide the actual latency spent using CPU by the *optimal latency* of executing this many cycles at the CPU frequency. Since part of the thread execution could be spent in synchronous IO operations, we only use CPU cycles and latency spent outside of these calls to compute CPU slowdown. If frequency scaling is enabled, we could use other existing performance counters to detect CPU contention [1].

Disk To estimate disk slowdown, we use a subset of disk IO operation types that we monitor, in particular, `reads` and `sync`s. For example, given a time interval with n `sync`s and b bytes written during these operations, we use a simple disk model that assumes a single seek with duration T_s for each `sync`, followed by data transfer at full disk bandwidth B. We thus estimate the optimal latency as $l = nT_s + b/B$ and slowdown as $s = t/l$, where t is the total time spent in `sync` operations. To deal with disk caching, buffering, and readahead, we only count as seeks the operations that took longer than a certain threshold, *e.g.*, 5ms. We use similar logic for reads and to estimate the load of each workflow.

Network The load of a workflow on a network link is proportional to the number of bytes transferred by that workflow. We ignore data sent over the loopback interface by checking remote address when the connection is set up, inside our AspectJ instrumentation. We currently do not measure the actual network latency and thus estimate the network slowdown based on its utilization by treating it as a M/M/1 queue. Thus a link with utilization u has a slowdown $1 + u/(1-u)$. It is feasible to extend Retro by encoding a model of the network (topology, bandwidths, and round trip times), and network flow parameters (source, destination, number of bytes), to estimate the network flow latency with no congestion [30]. Comparing this no-congestion estimate with measured latency could be used to compute network slowdown.

Thread pool The load of a workflow on a thread pool is proportional to the total amount of time it was using threads in this pool. Since we explicitly measure queuing and service time of a thread pool operation, we can directly compute the slowdown as total execution time (queuing plus service) divided by service time.

Locks A write lock behaves similarly to a thread pool with a single thread, and we explicitly measure the queuing time of a lock operation and the time the thread was holding the lock. Slowdown is thus the total latency of lock operation (from requesting the lock until release) divided by the time actually holding the lock.

Load of a read-write lock depends on the number of read and write operations, for how long they hold the lock, and the exact lock implementation. While there has been previous work on modeling locks using queues [24, 22, 32], none of them exactly match the ReentrantReadWriteLock used in HDFS. Instead, we approximate the capacity or throughput of a lock, $T(f,w,r)$, in a simple benchmark using three workflow parameters: fraction of write locks f, and average duration of write and read locks w and r. See Figure 4 for a subset of the measured values; notice that the throughput is nonlinear and non-monotonic. We use trilinear interpolation [23] to predict throughput for values not directly measured. Given a workflow with characteristic (f,w,r) and current lock throughput t, we estimate its load on the lock as

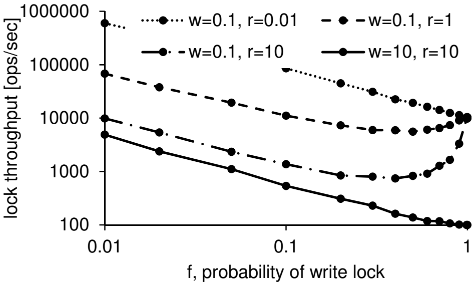

Figure 4: The throughput of Java ReentrantReadWriteLock (y-axis) as a function of three parameters: probability of a write lock operation (x-axis), average duration of read and write locks (see legend, time in milliseconds).

$t/T(f,r,w)$. E.g., a workflow making 1000 lock requests a second with its estimated max throughput of 5000 operations a second, would have a load of 0.2.

4.3 Coordinated throttling

Retro is designed to support multiple scheduling schemes, such as various queue schedulers or priority queues. In the current implementation of Retro, each control point is a *per-workflow distributed token bucket*. Threads can request tokens from the current workflow's token bucket, blocking until available. Queues can delay a request from being dequeued until sufficient tokens are available in the corresponding workflow's bucket. For a particular control point and workflow, a policy can set a rate limit R, which is then split (behind the scenes) across all point instances proportionally to the observed throughput. Retro keeps track of new control point instances coming and going – *e.g.*, mappers starting and finishing – and properly distributes the specified limit across them.

So long as each request executes a bounded amount of work, even using a single control point at the entrance to the system is enough for Retro to enforce usage of individual workflows. However, as described in Section 3.1, requests have to periodically pass through control points to guarantee fast convergence of allocation policies. Even without any control points in the system, each resource reports how many times it has been used by a particular workflow. For example, loading a single HDFS block of 64MB would result in approximately 1000 requests to the disk, each reading 64kB of data. These statistics help developers identify blocks of code where requests execute large amount of work and where adding control points helps break down execution and significantly improves convergence of control policies.

In the Hadoop stack, we added several points: in the HDFS NameNode and HBase RegionServer RPC queues, in the HDFS DataNode block sender and receiver, in the Yarn NodeManager, and in the MapReduce mappers when writing to the local disk. Each of these points has a number of instances equal to the number of processes of the particular type.

```
1   // identify slowest resource
2   S = r in resources() with max slowdown(r)
3   foreach w in workflows()
4     demand[w] = load(S, w)
5     capacity += $(1-\alpha)$*demand[w]
6
7   fair = MaxMinFairness(capacity, demand)
8
9   foreach w in workflows()
10    if (slowdown(S) > T
11        && fair[w] < demand[w]) // throttle
12      factor = fair[w] / demand[w]
13    else // probe for more demand
14      factor = (1 + $\beta$)
15
16    foreach p in points()
17      set_rate(p, w, factor*get_rate(p, w))
```

Algorithm 1: BFAIR policy, see §5.1.

Notice that we do not need to throttle directly on resource R to enforce resource limits on R. Assume that a workflow is achieving throughput of N_p at point p and has load L_R on R. By setting a throttling rate of αN_p for all points, we will indirectly control the load on R to αL_R.

5 Policies

This section describes three targeted reactive resource-management policies that we used to evaluate Retro. Specifically, these policies enforce fairness on the bottleneck resource (§5.1), dominant-resource fairness (§5.2), and end-to-end latency SLOs (§5.3). All of these polices are system-agnostic, resource-agnostic, and can be concisely stated in a few lines of code. These are not the only policies that could be implemented on top of Retro; in fact, we believe that the Retro abstractions allow developers to write more complex policies that consider a combination of fairness and latency, together with other metrics, such as throughput, workflow priorities, or deadlines.

5.1 BFAIR policy

The BFAIR policy provides bottleneck fairness [13, 12]; *i.e.*, if a resource is overloaded, the policy reduces the total load on this resource while ensuring max-min fairness for workflows that use this resource. This policy can be used to throttle aggressive workflows or to provide DoS protection. It provides coarse-grained performance isolation, since workflows are guaranteed a fair-share of the bottlenecked resource.

The policy, described in Algorithm 1, first identifies the slowest resource S in the system according to the slowdown measure (line 2). Then, the policy runs the max-min fairness algorithm with demands estimated by the current load of workflows (line 4) and resource capacity estimated by the total demand reduced by $1-\alpha$ to relieve the bottleneck if any (line 5).

The policy considers S to be bottlenecked if its slowdown is greater than a policy-specific threshold T. If this is the case and the fair share fair[w] of workflow w is

```
1  // estimate resource demands
2  foreach w in workflows()
3    foreach r in resources()
4      demand[r,w] = $(1+\alpha)$*load(r,w)
5
6  // update capacity estimates
7  cap = current capacity estimates
8  foreach r in resources()
9    tot_load = $\Sigma_w$load(r,w)
10   if(slowdown(r) > $T_r$) //reduce estimate
11     cap[r] = min(cap[r], tot_load);
12   else // probe for more capacity
13     cap[r] = max(($1+\beta$)*cap[r], tot_load);
14
15 share = DRF(demand, cap)
16 foreach w in workflows()
17   if (share[w] >= 1) continue
18   foreach p in points()
19     set_rate(p, w, share[w]*get_rate(p,w))
```

Algorithm 2: RDRF policy, see §5.2.

smaller than its current load (line 11), the policy throttles the rate by a factor of `fair[w]/demand[w]`. Here, the policy assumes a linear relationship between throughput at control points and the load on resources. If either the resource is not bottlenecked or if a workflow is not meeting its fair share (line 13), the policy increases the throttling rate by a factor of $1+\beta$ to probe for more demand.

Notice that this policy performs *coordinated* throttling of the workflow across all the control points; by reducing the rate proportionally on each point, we quickly reduce the load of the workflow on all resources. Parameters α and β control how aggressively the policy reacts to overloaded resources and underutilized workflows respectively. Notice that this policy will throttle only if there is a bottleneck in the system; we can change the definition of a bottleneck using the parameter `T`.

5.2 RDRF policy

Dominant resource fairness (DRF) [13] is a multi-resource fairness algorithm with many desirable properties. The RDRF policy (Algorithm 2) calls the original DRF function at line 15 which requires the current resource demands and capacities of all resources. In a general distributed system, we cannot directly measure the *actual resource demand* of a workflow, but only its current load on a resource. A workflow might not be able to meet its demand due to bottlenecks in the system.

The RDRF policy overcomes this problem by being reactive: making incremental changes and reacting to how the system responds to these changes. At any instant, the policy conservatively assumes that each workflow can increase its current demand by a factor of α (line 4). This increased allocation provides room for bottlenecked workflows to increase the load on resources.

Similarly, the policy uses the slowdown measure to estimate capacity. At line 10, when the current slowdown exceeds a resource-specific threshold, the policy reduces its capacity estimate to the current load. On the other hand, if the slowdown is within the threshold (line 12) and the current capacity estimate is lower than the current load, the policy increases the capacity estimate by a factor of β to probe for more capacity.

```
1  foreach w in H
2    miss(w) = latency(w) / target_lat(w)
3  h = w in H with max miss(w)
4
5  foreach l in L // compute gradients
6    g[l] = $\Sigma_r$ (latency(h,r) * log(slowdown(r))
7              * load(r,l) / $\Sigma_w$ load(r,w))
8
9  foreach l in L // normalize gradients
10   g[l] /= $\Sigma_k$g[k]
11
12 foreach l in L
13   if(miss(h) > 1) // throttle
14     factor = 1-$\alpha$*(miss(h)-1)*g[l]
15   else // relax
16     factor = (1 + $\beta$)
17
18   foreach p in points()
19     set_rate(p, l, factor*get_rate(p, l))
```

Algorithm 3: LATENCYSLO policy, see §5.3.

Given estimates of demand and capacity, the DRF() function returns `share[w]`, the fraction of `w`'s demand that was allocated based on dominant-resource fairness. If `share[w]` < 1, we throttle `w` at each point `p` proportionally to its current throughput at `p`.

5.3 LATENCYSLO policy

In the LATENCYSLO policy, we have a set of high-priority workflows H with a specified target latency SLO (service-level objective). Let L (low-priority) be the remaining workflows. The goal of the policy is to achieve the highest throughput for L, while meeting the latency targets for H. We assume the system has enough capacity to meet the SLOs for H in the absence of the workflows L; in other words, it is not necessary to throttle H. To maximize throughput, we want to throttle workflows in L as little as possible; *e.g.*, if a workflow in L is not using an overloaded resource, it should not be throttled.

Consider a workflow h in H that is missing its target latency. If multiple such workflows exist, the policy choses the one with the maximum `miss` ratio (line 3). Let t_w be the current request rate of workflow `w` and consider a possible change of this rate to $t_w * f_w$. The resulting latency l_h of h is some (nonlinear) function of the relative workflow rates f_w of all workflows. The LATENCYSLO computes an approximate gradient of l_h with respect to f_w and uses the gradient to move the throttling rates in the right direction. Based on the system response, the policy repeats this process until all latency targets are met.

We derive an approximation of l_h which results in an intuitive throttling policy. Consider a resource r with a current slowdown of S_r, load $D_{w,r}$ for workflow w, and total load $D_r = \sum_w D_{w,r}$. If $L_{h,r}$ is the current latency of h at r, the baseline latency is $L_{h,r}/S_r$ when there is no load at r, by the definition of slowdown. We model the latency of h at r, $l_{h,r}$ as an exponential function of the load d_r that satisfies the current ($d_r = D_r$) and baseline ($d_r = 0$) latencies, and obtain $l_{h,r} = L_{h,r} * S_r^{d_r/D_r - 1}$. Finally, we model the latency of h, $l_h = \sum_r l_{h,r}$ as the sum of latencies across all resources in the system.

Assuming that a fractional change in a workflow's request rate results in the same fractional change in its load on the resources, we have $d_r = \sum_w D_{w,r} * f_w$. The gradient of $l_{h,r}$ with respect to f_w at $d_r = D_r$ is $\partial l_{h,r} / \partial f_w = L_{h,r} * \log S_r * D_{w,r}/D_r$. This is a very intuitive result: the impact of workflow w on the latency of h is high if it has a high resource share, $D_{w,r}/D_r$, on a resource with high slowdown, $\log S_r$, and where workflow h spends a lot of time, $L_{h,r}$.

Algorithm 3 uses this formula for the gradient calculation (line 6). The policy throttles workflows in L based on the normalized gradients after dampening by a factor α to ensure that the policy only takes small steps. If all workflows in H meet their latency guarantees, the policy uses this opportunity to relax the throttling by a factor β.

6 Evaluation

In this section we evaluate Retro in the context of the Hadoop stack. We have instrumented five open-source systems – HDFS, Yarn, MapReduce, HBase, and ZooKeeper – that are widely used in production today. We use a wide variety of workflows, which are based on real-world traces, widely-used benchmarks, and other workloads known to cause resource overload in production systems.

Our evaluation shows that Retro addresses the challenges in §2.2 when applied simultaneously to all these stack components. In particular, we show that Retro:

- applies coordinated throttling to achieve bottleneck and dominant resource fairness (§6.1 and §6.3);
- applies policies to application-level resources, resources shared between multiple processes, and resources with multiple instances across the cluster;
- guarantees end-to-end latency in the face of workloads contending on different resources, uniformly for client and system maintenance workflows (§6.2);
- is scalable and has very low developer and execution overhead (§6.4);
- throttles efficiently: it correctly detects bottlenecked resources and applies *targeted throttling* to the relevant workflows and control points.

We do not directly compare to other policies, since to our knowledge, no previous systems offer this rich source of per-workflow and per-resource data. Many of previous policies, such as Cake [44], could be directly implemented on top of Retro.

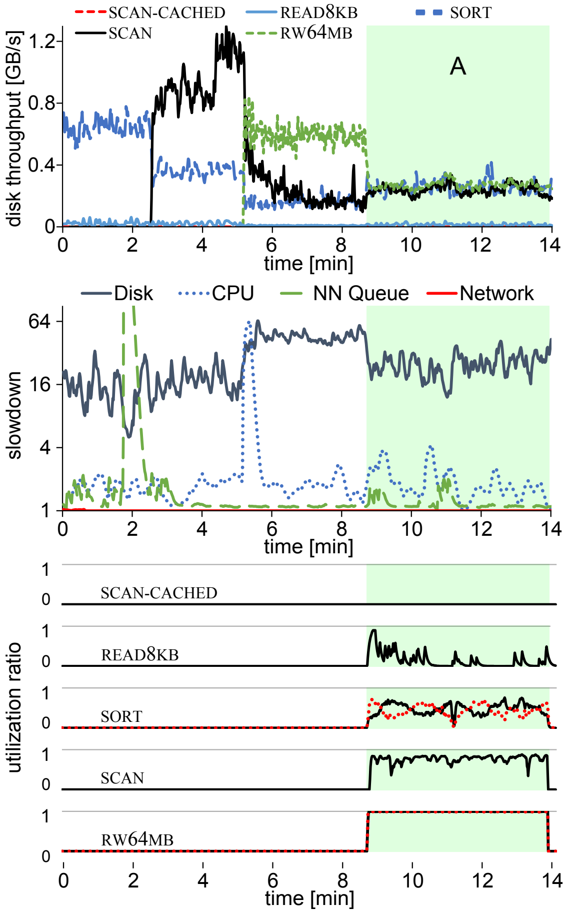

Figure 5: BFAIR policy as described in the text.

6.1 BFAIR in Hadoop stack

In Figure 5, we demonstrate the BFAIR policy successfully throttling aggressive workflows without negatively affecting the throughput of other workflows. The three *major* workflows are: SORT, a MapReduce sort job; RW64MB, 100 HDFS clients reading and writing 64MB files with a 50/50 split; and SCAN, 100 HBase clients scanning large tables. These workflows bottleneck on the disk on the worker machines. The two *minor* workflows are: READ8KB, 32 clients reading 8kB files from HDFS; and SCAN-CACHED, 32 clients scanning tables in HBase that are mostly *cached* in the RegionServers. We perform this experiment on a 32-node deployment of Windows Azure virtual machines; one node runs the Retro controller, one node runs HDFS NameNode, Yarn, ZooKeeper, and HBase RegionServer, the other thirty are used as Hadoop workers. Each VM is a Standard_A4 instance with 8 cores, 14GB RAM and a 600GB data disk, connected by a 1Gbps network.

At the beginning of the experiment, we start READ8KB, SCAN-CACHED, and SORT together, and delay start of SCAN and RW64MB. Figure 5(top) shows the disk throughput achieved by each workflow; notice how the throughput changes as different workflows start, for ex-

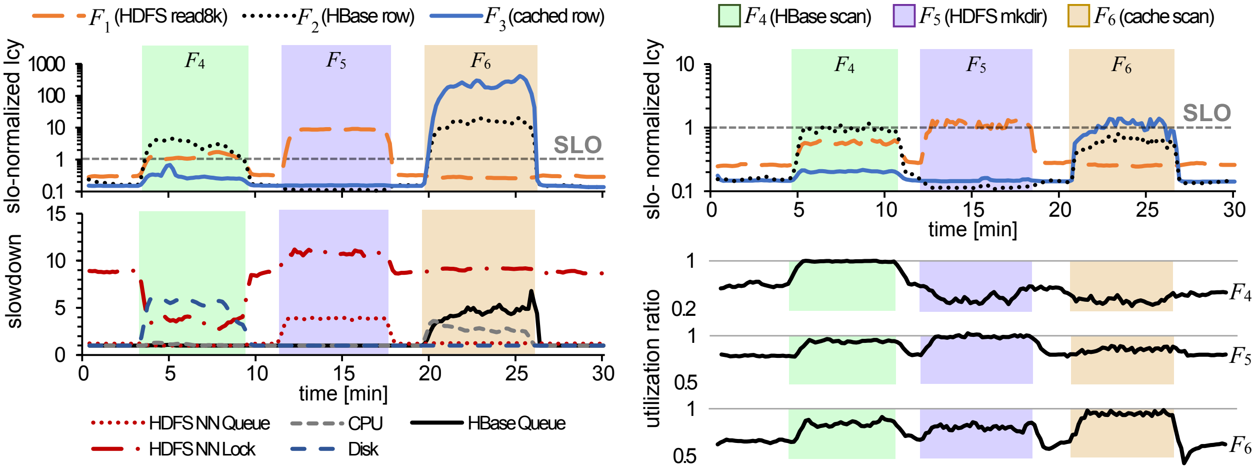

Figure 6: LatencySLO policy as described in text. Top-left figure shows high priority workflow latencies without LatencySLO. Bottom-left figure shows resource slowdown during experiment. Top-right figure shows high priority workflow latencies with LatencySLO. Bottom-right sparklines show control point utilizations for background workflows.

ample, throughput of SORT drops from 750MB/sec to 100MB/sec. Figure 5(center) shows the slowdown of a few different resources. Disk is the only constantly overloaded resource, reaching slowdown of up to 60. While slowdown of other resources also occasionally spikes, this happens only due to workload burstiness. In Figure 5(bottom), we show sparklines of the workflow *utilization ratios* – the achieved throughput relative to the allocated rate at a particular control point. A ratio of 1 means that the workflow is being actively rate-limited; a ratio of 0 means that the workflow is never rate-limited. For SORT, we show ratios at two control points: the DN BlockSender (black, used by mapper to read data from the DN) and mapper output (dashed red, used by mapper to write its output to local disk). For RW64MB, we show ratios at two control points: the DN BlockSender (black, used to read data from HDFS DNs) and the DN BlockReceiver (dashed red, used to write data to HDFS DNs).

In phase A we enable the BFair with overload threshold T=25. Quickly, the disk throughput of the three major workflows equalizes at about 300MB/sec, thus achieving fairness on the bottlenecked resource. Also, the disk slowdown fluctuates at around 25 (navy blue line in the slowdown graph) because the policy starts throttling the major workflows.

The utilization ratio sparklines provide further insight. RW64MB is the most aggressive workflow and consequently it is fully throttled (ratio of 1) at all of the control points. While not as aggressive, SCAN is also throttled though less. Depending on the phase of the map-reduce computation, we throttle SORT while reading input (black) and/or when writing output (red dashed). Finally, as expected, the two minor workflows are not throttled as much, or at all, because the fairness allocates their full demand. Furthermore, SCAN-CACHED is completely unthrottled as it has no disk utilization.

These results highlight how Retro enables coordinated and targeted throttling of workloads. No other system we are aware of would achieve these results, as Retro coordinates the same resource through different control points – for example, disk is controlled not only by HDFS block transfer (used by SCAN, RW64MB, READ8KB and the job input to SORT), but also by the SORT mapper output that accesses disk directly, bypassing HDFS. Retro only throttles the relevant workloads, leaving the small read and scan workloads mostly alone.

6.2 LatencySLO

We demonstrate that the LatencySLO policy can enforce a) end-to-end latency SLOs across multiple workflows and systems, and b) SLOs for both front-end clients and background tasks. We perform these experiments on an 8-node cluster; one node runs the Retro controller, one node runs HDFS NameNode, Yarn, ZK, and HBase Master, the other 6 are used as Hadoop workers and HBase RegionServers.

Enforcing multiple guarantees In this experiment we simultaneously enforce SLOs in HBase and HDFS for three high priority workflows with intermittently aggressive background workflows. The three high priority workflows are: F_1 randomly reads 8kB from HDFS with 500ms SLO, F_2 randomly reads 1 row from a small table cached by HBase with 25ms SLO, and F_3 randomly reads 1 row from a large HBase table with 250ms SLO. The background workflows are: F_4 submits 400-row HBase table scans, F_5 creates directories in HDFS, and F_6 submits 400-row HBase table scans of a cached HBase table.

Figure 6(top-left) demonstrates the request latencies of the three high priority workflows, normalized to their SLOs. During each of the three phases of the experiment, a background workflow temporarily increases its request

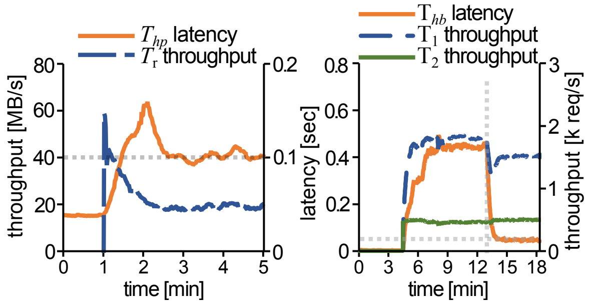

Figure 7: LATENCYSLO rate-limits replication to enforce a 100ms SLO for T_{hp} (left). LATENCYSLO enforces a 50ms latency for heartbeats (right).

rate, affecting the latency of the high priority workflows. In the first stage, F_4 increases its load and F_1 and F_2 miss their SLO. In the second stage, F_5 increases its load and F_1 misses its SLO by a factor of 10. In the last stage, F_6 increases its load and F_2 and F_3 miss their SLOs by factors of 10 and 500 respectively. Figure 6(bottom-left), shows the slowdown of different resources as the experiment progresses: at first F_4 table scans cause disk slowdown, then F_5 causes HDFS NameNode lock and NameNode queue slowdown, and finally F_6 causes CPU and HBase queue slowdown as its data is cached.

We repeat the experiment using LATENCYSLO to enforce the SLOs of F_1, F_2 and F_3. Figure 6(top-right) shows that the policy successfully maintains the SLOs by throttling the background workflows at a number of control points within HDFS and HBase. Figure 6(bottom-right) shows the sparklines of the workflow *utilization ratios* – the achieved throughput relative to the allocated rate at a particular control point, similar to Figure 5. We see that LATENCYSLO only rate-limits the background workflows during their specific overload phases.

These results highlight how LATENCYSLO selectively throttles workloads based on their contribution to the SLO violation. Retro can enforce SLOs for multiple workflows across software and hardware resources simultaneously.

Background workflows Thanks to the workflow abstraction, LATENCYSLO is equally applicable to providing guarantees for high priority background tasks, such as heartbeats, or to protecting high priority workflows from aggressive background tasks such as data replication.

Figure 7(right) demonstrates the effect of two workflows T_1 and T_2 on the latency of datanode heartbeats, T_{hb}. The heartbeat latency increases from 4ms to about 450ms when T_1 and T_2 start renaming files and listing directories, respectively, causing increased load the HDFS namesystem lock. Whilst T_{hb} and T_2 only require read locks, T_1 requires write locks to update the filesystem, thus blocking heartbeats. When we start SLO enforcement at t=13, the policy identifies T_1 as the cause of slowdown, throttles it at the NameNode RPC queue, and achieves the heartbeat SLO.

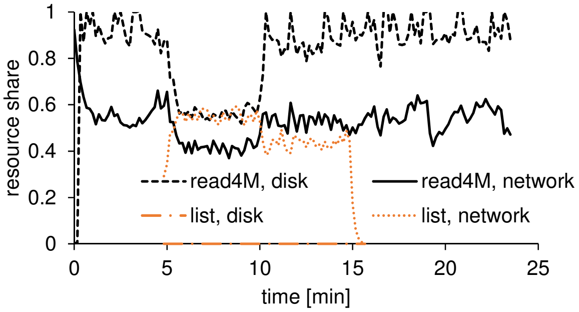

Figure 8: Resource share for experiment described in §6.3.

In Figure 7(left), LATENCYSLO rate-limits low-priority background replication T_r, to provide guaranteed latency to high priority workflow T_{hp} submitting 8kB read requests with 100ms SLO. At t=1, we manually trigger replication of a large number of HDFS blocks; subsequently, LATENCYSLO rate-limits T_r. High-priority replication (single remaining replica) could use a separate workflow ID to avoid throttling.

6.3 RDRF in HDFS

To demonstrate RDRF (Figure 8), we run an experiment with two workflows – READ4M with 50 clients reading 4MB files, and LIST with 5 clients listing 1000 files in a directory – accessing the HDFS cluster remotely sharing a 1Gbps network link. The dominant resource for READ4M is disk and for LIST it is the network, since it is reading large amounts of data from the memory of the NameNode.

We start READ4M at t=0 and add LIST at t=5, with sharing weights of 1. Between time 5 and 10, RDRF throttles READ4M to achieve equal dominant shares across both of these workflows (60% on disk and network). After increasing the weight of READ4M to 2 at t=10, the dominant shares change to 80% and 40%, respectively.

Despite knowing neither the demands of each workflow, nor the capacity of each resource, RDRF successfully allocates each workflow the fair share of its dominant resource. The experiment demonstrates how slowdown is viable as a proxy for resource capacity, and coupled with reactive policies, enables us to overcome some limitations of an existing resource fairness technique.

6.4 Overhead and scalability of Retro

Retro propagates a workflow ID (3 bytes) along the execution path of a request, incurring up to 80ns of overhead (see Table 2) 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-workflow report.

To estimate the impact of Retro on throughput and end-to-end latency, we benchmark HDFS and HDFS instrumented with Retro using requests derived from the

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

Table 2: Costs of Retro instrumentation

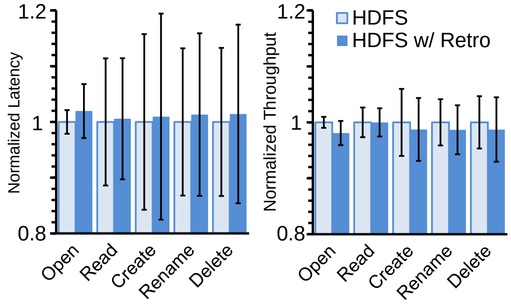

Figure 9: Normalized latency (left) and throughput (right) for HDFS NameNode benchmark operations along with error bars showing one standard deviation.

HDFS NNBench benchmark. See Figure 9 for throughput and end-to-end latency for five requests types. *Open* opens a file for reading; *Read* reads 8kB of data from a file; *Create* creates a file for writing; *Rename* renames an existing file and *Delete* deletes the file from the name system (and triggers an asynchronous block delete). Of the request types, *Read* is a DataNode operation and the others are NameNode operations. In all cases, latency increases by approximately 1-2%, and throughput drops by a similar 1-2%. Variance in latency and throughput increases slightly in HDFS instrumented with Retro. These overheads could be further significantly reduced by *sampling*, *i.e.*, tracing only a subset of requests or operations.

We evaluate Retro's ability to scale beyond the cluster sizes presented thus far with an 200-VM experiment on Windows Azure (Standard_A2 instances). Figure 10 shows slowdown and aggregate disk throughput for four workflows when BFair is activated (at t=1.5) and per-workflow weights are adjusted (at t=4). Each workflow ran a mix of 64MB HDFS reads and writes, with 800, 1200, 1600, and 2000 closed-loop clients respectively. Before the policy is activated we observe the expected imbalance in disk throughput caused by the differing number of clients in each workflow. When the policy is activated at t=1.5, the workflows quickly converge to an equal share of disk throughput, and the slowdown decreases to the target of 50. At t=4, two of the clients are given a weight of 2 and the policy quickly establishes the new fair share.

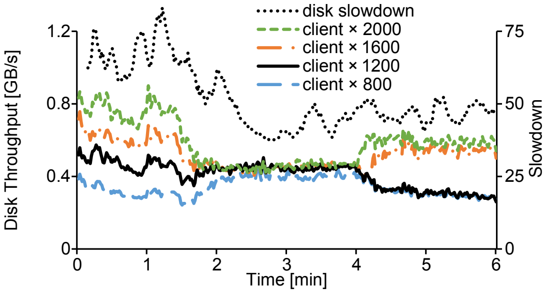

Figure 10: Retro's BFair policy on a 200-node cluster with four workflows and overloaded disks. BFair is enabled at t=1.5 with a target slowdown of 50; client weights are adjusted at t=4.

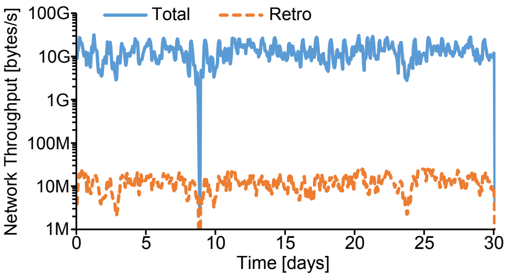

Figure 11: Total network throughput for a several-hundred node production Hadoop cluster and network throughput of Retro, calculated from 1 month of traces. Retro's bandwidth requirements are on average 0.1% of the total throughput.

We evaluate the scalability of Retro's central controller in terms of its ability to process resource reports. In a benchmark where each report contains resource usage for 1000 workflows, the controller can process on the order of 10,000 reports per second. Assuming 10 resources per machine, the controller could thus support up to 1000 machines. In this setup, each machine would use about 600kB/sec of network bandwidth to send the reports. Figure 11 shows calculated network overhead that would be imposed by Retro on a production Hadoop cluster comprising several hundred nodes over a period of a month. We calculate the network traffic that would be generated by Retro based on traces from this production cluster. The figure shows that Retro would account for an average of 0.1% of the network traffic present. Furthermore, since Retro aggregation only computes sums and averages, we can aggregate hierarchically (*e.g.* inside each machine and rack), further reducing the required network bandwidth and thereby supporting much larger deployments.

Whilst Retro requires manual developer intervention to propagate workflow IDs across network boundaries and to verify correct behavior of Retro's automatic instrumentation, our experience shows that this requires little work. For example, instrumenting each of the five systems required only between 50 and 200 lines of code; for example to handle RPC messages. Instrumenting resource operations happens automatically through AspectJ.

7 Discussion

In Retro, we made the decision to implement both resource measurement and control points at the application level. While applying Retro in the OS, hypervisor, or device driver level could provide more accurate measurements and fine-granularity enforcement, our approach has the advantages of fast and pervasive deployment, and of not requiring specially built OS or drivers (we deployed Retro in both Windows and Linux environments). Retro's promising results indicate that OS's, and distributed systems in general, should provide mechanisms to facilitate the propagation of workflow IDs across their components.

Retro is extensible to handle *custom resources*. For example, in systems with row-level locking we cannot treat each lock/row as an individual resource because the number of resources might be unbounded. Instead, we could define each data partition as a logical resource, which would significantly reduce the number of resources in the system. ZooKeeper uses a custom request processing pipeline, which is not part of Java standard library. We treat ZooKeeper queues as custom resources and estimate their load and slowdown.

The current implementation of Retro has several limitations. First, some resources cannot be automatically *revoked* once a request has obtained them and have to be explicitly released by the system. For example, this applies to memory, sockets, or disk space. A developer could implement application-specific hooks that Retro could use to reclaim resources. Second, because the rates of distributed token buckets are updated only once a second, when workload is very variable, this might reduce the throughput of the system. Using different local schedulers, such as weighted fair queues [35] and reservations [15] would alleviate this problem.

8 Related work

In [26] we introduced the design principles behind Retro, as well as a preliminary implementation of resource measurement for HDFS. This paper presents a complete framework by adding the centralized controller, resource management policies, and distributed control points, evaluated across five different distributed systems.

Multi-resource scheduling Several research projects tackle multi-resource allocation, such as Cake [44], mClock [15], IOFlow [39], and SQLVM [27]. These frameworks are specific to particular systems, such as storage or relational databases. Retro improves on top of these by providing the workflow, resource, and control point abstractions, which allow it to handle a wide range of resources and system activities, and enforce policy decisions across the whole system.

Cake provides isolation between low-latency and high-throughput tenants using HBase and HDFS. However, it treats HDFS as a single resource, and cannot target specific resource bottlenecks and workflows that overload these resources. mClock is a disk IO scheduler that could be implemented as a Retro control point. IOFlow provides per-tenant guarantees for remote disk IO requests in datacenters but does not schedule other resources such as threadpools, CPU, and locks. SQLVM [27] provides isolation for CPU, disk IO, and memory for multiple relational databases deployed in a single machine, but does not deal with distributed scenarios.

In the data analytics domain, task schedulers such as Mesos [20], Yarn [42], or Sparrow [29] use an admission control approach to allocate individual tasks to machines. In these frameworks, each task passes through the scheduler before starting its execution, the scheduler can place it to an arbitrary machine in the cluster and after starting execution, the task is not scheduled any more. In typical distributed systems, requests do not pass through a single point of execution and routing of a request through the system is driven by complex internal logic. Finally, to achieve fine-grained control over resource consumption, requests have to be throttled *during* its execution, not only at the beginning. These frameworks thus do not directly apply to scheduling in general distributed systems. On the other hand, Retro requires no knowledge of internal design of the system and provides fine-grained throttling using control points on the request execution path.

End-to-end (resource) tracing Banga and Druschel addressed the mismatch between OS abstractions and the needs of resource accounting with resource containers [4], which, albeit in a single machine, aggregate resource usage orthogonally to processes, threads, or users. Our end-to-end propagation of workflow IDs shares mechanisms with taint tracking [28] and several causal tracing frameworks [5, 8, 9, 11, 34, 37, 40]. Retro does not, however, record causality or traces, but rather uses the workflow information to attribute resource usage. Whodunit [7] uses causal propagation to record timings between parts of a program, and provides a profile of where requests spent their time. Timecard [33] also propagates cumulative time information in the request path between a mobile web client and a server, and uses this in real time to speed up the processing of requests that are late. Retro, in contrast, records aggregate resource profiles by workflow and uses these to enforce flexible high-level policies.

9 Conclusion

Retro is a framework for implementing resource management policies in multi-tenant distributed systems. Retro tackles important challenges and provides key abstractions that enable a separation between resource-management policies and mechanisms. It requires low developer effort, and is lightweight enough to be run in production. We demonstrate the applicability of Retro to key components of the Hadoop stack and develop and evaluate three targeted and reactive policies for achieving fairness and latency targets. These policies are system-agnostic, resource-agnostic, and uniformly treat all system activities, including background management tasks. To the best of our knowledge, Retro is the first framework to do so. We plan to extend the control points to provide fair scheduling, prioritization, and load balancing.

References

[1] Intel Performance Counter Monitor – A better way to measure CPU utilization. `http://intel.ly/1C23e67`.

[2] F. Akgul. *ZeroMQ*. Packt Publishing, 2013.

[3] Amazon web services. `http://aws.amazon.com/`.

[4] 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.

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

[6] 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 *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[7] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional Profiling for Multi-Tier Applications. In *EuroSys'07*, Lisbon, Portugal, March 2007.

[8] A. Chanda, K. Elmeleegy, A. L. Cox, and W. Zwaenepoel. Causeway: System support for controlling and analyzing the execution of multi-tier applications. In *Proc. Middleware 2005*, pages 42–59, November 2005.

[9] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem Determination in Large, Dynamic, Internet Services. In *Proc. International Conference on Dependable Systems and Networks*, 2002.

[10] T. Do, H. S. Gunawi, T. Do, T. Harter, Y. Liu, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. The case for limping-hardware tolerant clouds. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2013.

[11] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation*, NSDI'07, Berkeley, CA, USA, 2007. USENIX Association.

[12] 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.

[13] 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*, 2011.

[14] Google Protocol Buffers. `http://code.google.com/p/protobuf/`.

[15] A. Gulati, A. Merchant, and P. J. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In R. H. Arpaci-Dusseau and B. Chen, editors, *Proceedings of OSDI*, pages 437–450. USENIX Association, 2010.

[16] 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.

[17] HBase. `http://hbase.apache.org`.

[18] HDFS-4183. `http://bit.ly/1l4uWbu`.

[19] HDFS API. `http://bit.ly/1cxFTD9`.

[20] 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. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, 2010.

[22] T. Johnson. Approximate analysis of reader/writer queues. *IEEE Trans. Softw. Eng.*, 21(3):209–218, Mar. 1995.

[23] H. Kang. *Computational Color Technology*. Press Monographs. Society of Photo Optical, 2006.

[24] S.-I. Kang and H.-K. Lee. Analysis and solution of non-preemptive policies for scheduling readers and writers. *SIGOPS Oper. Syst. Rev.*, 32(3):30–50, July 1998.

[25] 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.

[26] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. Towards general-purpose resource management in shared cloud services. In *10th Workshop on Hot Topics in System Dependability (HotDep 14)*, Broomfield, CO, Oct. 2014. USENIX Association.

[27] 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.

[28] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, Feb. 2005.

[29] 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.

[30] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *ACM SIGCOMM Computer Communication Review*, volume 28(4), pages 303–314. ACM, 1998.

[31] A. K. Parekh and R. G. Gallagher. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Transactions on Networking (TON)*, 2(2):137–150, 1994.

[32] L. C. Puryear and V. G. Kulkarni. Comparison of stability and queueing times for reader-writer queues. *Perform. Eval.*, 30(4):195–215, 1997.

[33] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *SOSP '13*, pages 85–100, New York, NY, USA, 2013. ACM.

[34] 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.

[35] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round-robin. *Networking, IEEE/ACM Transactions on*, 4(3):375–385, 1996.

[36] 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.

[37] 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.

[38] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking (ToN)*, 6(5):611–624, 1998.

[39] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A Software-defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196. ACM, 2013.

[40] 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 Perform. Eval. Rev.*, 34(1):3–14, June 2006.

[41] 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*, pages 996 –1005, march 2010.

[42] 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. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[43] A. Wang. Personal communication.

[44] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: Enabling High-level SLOs on Shared