Radio+Tuner: A Tunable Distributed Object Store

Dorian J. Perkins * †, Curtis Yu †, and Harsha V. Madhyastha
University of California, Riverside

1 Problem Statement

The primary reason for diversity of distributed storage systems is that no single design for a distributed storage system is cost-optimal for meeting the latency and throughput SLOs in all possible workloads. In other words, a distributed object store that satisfies the SLOs for a particular workload at minimum cost (in comparison to other object stores) will incur higher costs than alternate storage systems on other workloads.

Therefore, an application administrator today is faced with the onerous task of choosing from the large variety of available object stores; an incorrect choice may significantly inflate costs necessary to meet performance SLOs. Furthermore, as new workloads emerge, none of the existing systems may be a good match, thus necessitating the development of new object stores. This is because existing systems are inflexible, having assumptions of their target workload are baked into their implementation, and making it unclear how any particular system should be modified in order for it to be well-suited for a workload other than that for which it was designed. While any existing system can likely meet an application's performance goals with infinite resources, we believe this is the wrong approach.

2 Motivation

Our work is motivated by the inter-dependence between the configuration in which data is stored and the hardware necessary to meet performance SLOs. We demonstrate this using three object stores—HDFS, Voldemort, and Haystack—that have been designed for disparate workloads. HDFS, which mimics the design of GFS, is well-suited for storing and processing large objects at high throughput. Voldemort mimics the design of Dynamo, which was designed to store shopping carts on Amazon and to serve GETs/PUTs on these small objects with low latencies. Finally, Haystack is designed to store photos uploaded to Facebook, and is optimized to enable low-latency reads on write-once read-many objects. Since Haystack is not available for public use, we use Bitcask to build Haystack*, an imitation of Haystack's design.

We test the performance offered by these three object stores on the three workloads that are the target workloads for these systems. Though these workloads are relatively simplistic (e.g., we consider no skew across objects either in terms of size or read/write rates), these suffice for our purpose of demonstrating the variance in a particular system's performance across workloads. For each (object store, workload) pair, we plot the ratio of the GET latency SLO for that workload to the corresponding percentile of GET latencies measured when using that object store to serve the workload (Figure 1). For example, the bar for Voldemort on *Workload2* is the ratio of 100 ms (the SLO in *Workload2*) to the 99th percentile GET latency when using Voldemort for *Workload2*.

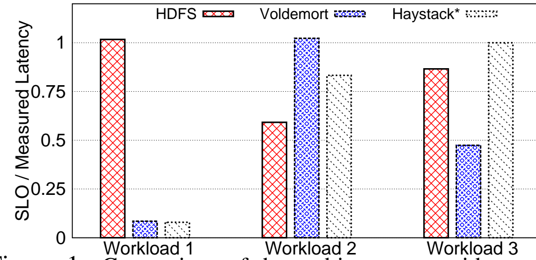

Figure 1: Comparison of three object stores with respect to their ability to meet the SLOs in three different workloads.

Overall, though the cost of hardware was the same in the deployments of all three object stores, the ability to meet the performance SLOs associated with a particular workload varies significantly across the object stores. In all cases where the SLOs were not met, a larger storage cluster would be necessary, thus showing that the wrong choice of storage configuration can inflate the cost necessary to meet performance SLOs.

We observe that existing object stores are inflexible because they choose to implement each of their components in one particular manner—that which is best suited for the system's target workload. As a result, though these systems expose several configuration parameters and tweaking these parameters can significantly vary performance, the range of parameters is limited.

3 Overview

Motivated by the inflexibility of existing object stores, we design a tunable object store, *Radio*, and its associated configuration engine, *Tuner*. At this time, *Radio* offers a simple PUT/GET interface (Figure 2), but is extensible to other types of transactions.

3.1 *Radio* object store

In contrast to existing object stores, *Radio* makes no a priori assumptions of its input workload. *Radio* can be deployed in a range of configurations, and in any particular deployment scenario, one can *tune* it to that configuration which is well-suited for the target workload.

* Presenting author. † This author is a student.

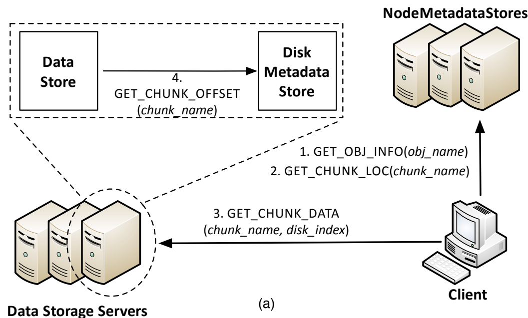

Figure 2: Interaction between *Radio*'s components to serve GETs

Radio's design is based on the principle of separating mechanism from policy. We partition *Radio* into the logical components that any object store must implement—storing metadata on how objects are split into chunks, storing metadata of where chunks are stored on disk, and storing the chunks themselves on disk. We permit several implementations of each component, with all implementations of a component exporting the same interface. Each of the implementations for any component presents a different trade-off between performance and cost. Thus, in any particular deployment scenario, we can select a specific implementation for each of *Radio*'s components, such that the chosen combination of implementations is well-suited for the target workload. Moreover, as new workloads emerge, we need only add new implementations for specific components rather than develop a new object store. Since new implementations of a component export the same interface, they can be paired with any one of the implementations of the other two components.

3.2 Configuring *Radio* with *Tuner*

Though the *Radio* object store is tunable, choosing the right configuration for it manually will be onerous. For every deployment, it is impractical for one to choose the right configuration of *Radio* by testing each empirically.

Therefore, we develop *Tuner*—a configuration engine associated with the *Radio* object store. To select the appropriate configuration of *Radio* for any given deployment, *Tuner* takes as input characterizations of the 1) cluster hardware, 2) application's workload, 3) performance SLOs, and 4) available implementations of *Radio*'s components. Given these inputs, *Tuner* abstractly navigates the search space of all possible configurations of *Radio*. For each configuration, *Tuner* estimates the cost of hardware that will be necessary to meet the performance SLOs if *Radio* were deployed in that particular configuration. *Tuner* then picks that configuration with the lowest cost estimate.

4 Initial Results

Our initial results demonstrate that our implementation of *Radio* can adapt well to different workloads.

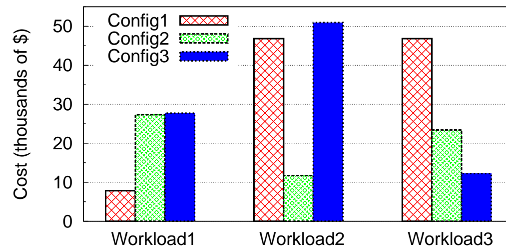

Figure 3: Comparison of cost to meet SLOs for different workloads with different *Radio* configurations.

4.1 Evaluation setup

Cluster. Our experiments are run on a cluster of 12 1U rack servers, each with two hyper-threaded quad-core Intel Xeon 2.26 GHz CPUs, 24 GB RAM, and eight 500 GB 7.2K RPM SAS drives. All servers are on a shared 10 GbE network. We deploy *Radio* on a subset of these servers and test it on several workloads.

Workloads. In our evaluations, we use three workloads with disparate characteristics, each well-suited for either HDFS, Dynamo, or Haystack. In all cases, we consider the storage capacity requirement to be 5 TB. We have implemented a custom workload generator that outputs a trace of GET/PUT requests in keeping with the workload's properties. When experimenting with any given workload, we play several of such traces in parallel from a separate set of 4 servers, which are also connected to the storage servers with 10 Gig Ethernet. The wide variance in these three workloads tests various facets of *Radio* and *Tuner*.

4.2 Configurability of *Radio*

We evaluate *Radio*'s adaptability to different workloads. For this, we provide each of our three example workloads, along with characterizations of our hardware, as input to *Tuner*. *Tuner* outputs the cost-effective configuration for each, where the ith configuration is the output for the ith workload. For each workload, we then deploy *Radio* in each of the three configurations, and gather measurements at different scales to determine the number of servers necessary to meet the workload's SLO. In some cases, our cluster of 12 servers is insufficient to meet the SLO.

Figure 3 plots, for every workload, the cost incurred in meeting the SLO with each of the considered configurations. For each workload, we see that the cost necessary to meet the SLO is significantly less with the *Radio* configuration chosen by *Tuner*. This is due to two reasons. First, when we keep the cost the same across all three configurations, the performance obtained with *Radio* configurations that are not well-suited for the particular workload is well below the SLO (figure omitted). Second, this reflects *Tuner*'s ability to accurately pick the most cost-effective configuration.