A Methodology for Understanding MapReduce Performance Under Diverse Workloads

Yanpei Chen
Archana Sulochana Ganapathi
Rean Griffith
Randy H. Katz

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-135
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-135.html

November 9, 2010

A Methodology for Understanding MapReduce Performance Under Diverse Workloads

ABSTRACT

MapReduce is a popular, but still insufficiently understood paradigm for large-scale, distributed, data-intensive computation. The variety of MapReduce applications and deployment environments makes it difficult to model MapReduce performance and generalize design improvements. In this paper, we present a methodology to understand performance tradeoffs for MapReduce workloads. Using production workload traces from Facebook and Yahoo, we develop an empirical workload model and use it to generate and replay synthetic workloads. We demonstrate how to use this methodology to answer "what-if" questions pertaining to system size, data intensity and hardware/software configuration.

1. INTRODUCTION

The MapReduce programming paradigm is becoming increasingly popular for parallelizing large-scale data processing for many applications, including web search, financial modeling, facial recognition, and data analytics [7]. Due to its popularity, much recent work seeks to extend the functionality of MapReduce or optimize its performance [20, 4, 19, 2, 9, 15, 11]. However, the diversity of MapReduce applications makes it difficult to develop performance improvements that are portable across different use cases. Optimizations for one environment may not be relevant for another. What is needed is a systematic way to evaluate whether the proposed improvements are specific to a particular workload on a given configuration, or more generally applicable.

In this paper, we propose a general performance measurement methodology for MapReduce that is workload independent. Figure 1 provides a graphical summary of our methodology. First, we collect traces from a production cluster, and extract an empirical description of workload features. Then, we generate synthetic workloads that capture representative behavior. We replay these synthetic workloads on small scale test clusters and measure the resulting performance. Finally, based on these measurements, we decide whether we can deploy the proposed improvements on a larger system configuration. Our key contributions are: (1) the identification of the performance-relevant workload features, (2) the development of an algorithm to generate synthetic workloads that scale up in data size and execution time, and (3) an extrapolation procedure that translates observed performance on a small scale test system to performance on a larger scale production system.

This methodology allows us to explore various "what-if" questions. For example, MapReduce developers can measure the performance improvements of tuning the cluster configurations, upgrading the cluster hardware, increasing the cluster size, or porting the cluster to newer versions of the MapReduce software. Likewise, MapReduce developers can anticipate the performance implications of workload changes, such as projected growth in job number and data size, or the multiplexing of different workloads on the same cluster. We can further assess the generality of proposed MapReduce optimizations by running different workloads using the improved system.

Developing such a methodology is difficult. Even if we have access to production clusters and workloads, the relevant workload features are not obvious. If we wrongly include some deployment specific characteristics as features, such as misconfigurations that lead to pathologically long running times, we may replicate system defects across our experiments. Also, it is not obvious how to scale the workload in size and in time, or how to replay the workload when the exact production data and production code is changing or unavailable. Without good knowledge about workload features, synthesis methods, and replay mechanisms, we would not know which subset of observed behavior is generalizeable across systems or across workloads.

Fortunately, MapReduce has a well-defined processing pattern. The input consists of key-value pairs, each passing through a map function, which generates intermediate key-value pairs. Each intermediate pair then passes through a reduce function, which outputs the final key-value pairs.

The map and reduce functions execute in parallel across the MapReduce cluster, with each node operating on a partition of the input and intermediate data. The intermediate data is often generated at one node but consumed at another, requiring a data shuffle across the network. We draw inspiration from past successes of developing a similar workload-independent measurement methodologies for Internet traffic

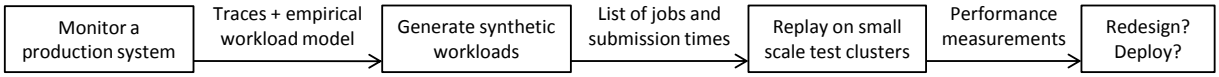

Figure 1: Workload model-synthesis-replay pipeline.

[13] and dynamic web content [16, 17, 3], two application domains with similarly well-defined processing patterns.

The roadmap for the rest of the paper is as follows. In Section 2, we compare two production MapReduce traces, one from Facebook, another from Yahoo. This analysis allows us to understand key workload features, such as data sizes and ratios, job arrival intensities and sequences, and the actual map and reduce functions. These workload features provide a logical description of "what processing is being done." We then outline our workload synthesis and replay methodology in Section 3. We identify and evaluate ways of scaling different workload dimensions, e.g., cluster size, hardware, data size, or workload duration. We further investigate the effects of replaying the same workload across different deployments when the desired production cluster, data, or code is unavailable. These experiements allow us to identify how to translate performance measurements across different MapReduce deployments and to evaluate what we lose by scaling or excluding specific features.

We further demonstrate how to apply this methodology for an example use case, in which the operator of a large scale MapReduce cluster measures the performance improvement of running the same workload on newer MapReduce versions (Section 4). The workload-level methodology leads to surprising performance metrics, such as job failure rate (we had assumed that all jobs would finish). We also observe unexpected results from some MapReduce schedulers, which perform far more effectively than previously reported [19].

Lastly, we discuss some related work on MapReduce performance optimizations (Section 5). Most of this work measures performance either using micro-benchmarks or through deployments on production clusters. We identify some ways that a workload-level methodology could yield new insights for these studies.

We hope the readers take away an understanding of why workload-level measurement methodology is necessary, what such a methodology involves, and how to implement and apply this methodology for their particular workloads.

2. WORKLOAD FEATURES

The goal of this section is to identify key workload features. We accomplish this goal by analyzing two traces from production MapReduce clusters at Facebook (FB) and Yahoo (YH). The FB trace comes from a 600-machine cluster, spans 6 months from May 2009 to October 2009, and contains roughly 1 million jobs. The YH trace comes from a cluster of approximately 2000 machines, covers three weeks in late February 2009 and early March 2009, and contains around 30,000 jobs. Both traces contain job sequences and hashed job names, job submission and completion times, data sizes for the input, shuffle and output stages, and the running time of map and reduce functions.

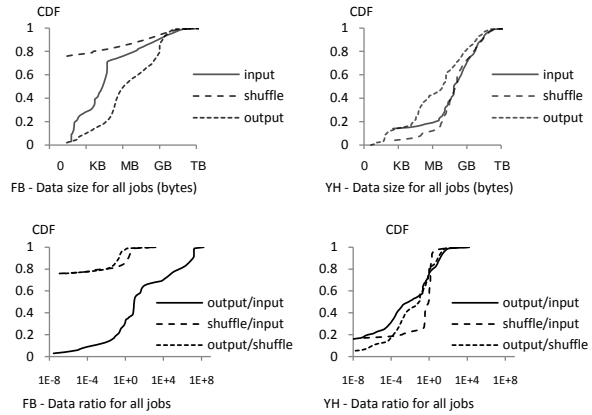

Figure 2: Workload data statistics.

We begin by comparing features that describe MapReduce data characteristics (Section 2.1). These characteristics differ greatly between workloads, and thus should be parameterized for each workload. We then examine the time varying pattern of job arrival rates and data intensities (Section 2.2). Even within the same workload, there is high peak-to-average ratio for these dimensions, requiring empirical models to capture such behavior. We further identify common jobs within each workload (Section 2.3). Both workloads contain many small jobs and few large jobs. However, the large jobs represent vastly different computations. Hence, our workload model should also capture the right proportion of different job types, and for each job type, the right dependency between various data dimensions. At the end of the section, we combine our key insights into an empirical workload model (Section 2.4), which forms the input to our synthetic workload generator.

2.1 Data characteristics

The MapReduce processing patterns operate on key-value pairs at input, shuffle (intermediate), and output stages. Hence, our first instinct is to compare data characteristics across FB and YH traces. Figure 2 shows aggregate data sizes and data ratios for each trace.

The input, shuffle, and output data sizes range from KBs to TBs. Within the same trace, the data size at different MapReduce stages follows vastly different distributions. Across the FB and YH traces, the same MapReduce stage has different distributions. We can thus surmise that the two MapReduce systems were performing very different computations on very different data sets. Additionally, many jobs in the FB trace have no shuffle stage - the map outputs are directly written to HDFS. Consequently the CDF of the shuffle data sizes have a high density at 0.

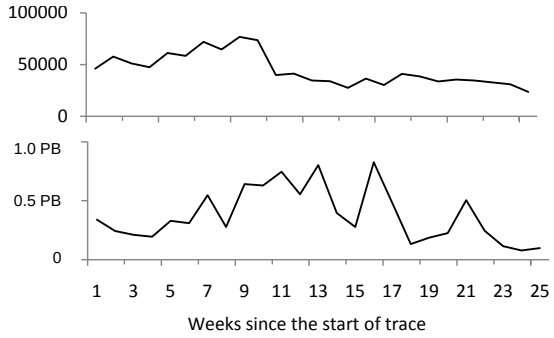

Figure 3: Weekly aggregate of job counts (top) and sum of map and reduce data size (below). FB trace.

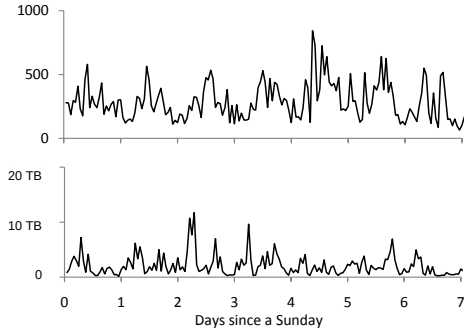

Figure 4: Hourly aggregate of job counts (top) and sum of map and reduce data size (below) over a randomly selected week. FB trace.

The data ratio between the output/input, shuffle/input, and output/shuffle stages also span several orders of magnitude, offering further evidence that the two MapReduce systems were performing very different types of work. Interestingly, there is very little density around 1 for the FB trace, indicating that most jobs are data expansions (ratio $>> 1$) or data compressions (ratio $<< 1$). The YH trace shows relatively more data transformations (ratio ≈ 1).

Implications for workload model: Data sizes at each of the input-shuffle-output stages need to be represented as a workload parameter. We further need to capture the appropriate data ratio between the different stages. The statistical distribution for each data size and data ratio is different across workloads, suggesting that we either fit a different analytical distribution for each workload, or parameterize the model using the empirical distributions.

2.2 Time varying characteristics

In addition to data characteristics, it is also important to look at job arrival patterns and data intensities, since they indicate "how much work" there is within a workload. We focus on the FB trace only for this analysis, because it offers insights over a longer time period.

Figure 3 shows weekly aggregates of job counts and the sum of input, shuffle, and output data sizes over the entire trace. There is no evident long term growth trend in the number of jobs or the sum of data size. Also, there is high variation in the sum data size but not in the number of jobs. We also see a discrete jump in the number of jobs in Week 11, which our

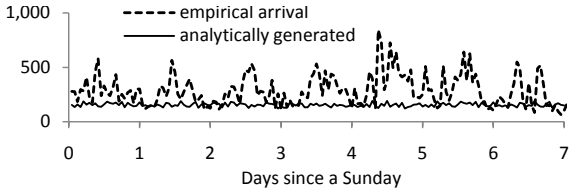

Figure 5: Comparison of empirical, hourly aggregate job arrival rates over a week against arrival rates generated by a random sampling arrival process.

Facebook collaborators clarified was due to a reorganization in cluster operations.

Figure 4 shows hourly aggregates of job counts and sum data sizes over a randomly selected week. We do not aggregate at scales below an hour as many jobs complete in tens of minutes to several hours. There is extremely high variation in both the number of jobs and the sum data size.

Also, the number of jobs show some diurnal patterns, peaking at mid-day and dropping at mid-night. We performed Fourier analysis on the hourly job counts to detect the existence of cycles in FB's job arrival rate. There are visible but weak signals at the 12 hour, daily, and weekly frequencies, mixed in with a high amount of noise. The YH trace is not long enough to enable similar analysis. Thus, we cannot conclude whether these diurnal patterns are generally applicable beyond the FB trace.

We also investigated whether a simple generative process could fit the job arrival patterns. We randomly sample the distribution of job inter-arrival times, computed over the entire trace, and generate a week-long job sequence. Figure 5 compares the hourly aggregate of this sequence with an empirical sequence. Clearly, this simple process cannot capture the full range of peak to average behavior.

Implications for workload model: Our workload parameter set must include time varying job arrival rate and data intensity. We need to capture any diurnal patterns, or at least the correct proportion of high, medium, and low activity periods. A simple random sampling of job arrival rates cannot capture peak to average behavior, suggesting that any synthetic workload generation processes need to be parameterized using the empirical job and data arrival rates.

2.3 Common jobs

Thus far, we have examined data characteristics (how large is each piece of work) and temporal characteristics (how much work there is at any point in time). A description of the MapReduce processing pattern also needs to capture *what* the work actually is. Our goal in this section is to investigate whether we can capture "most of the work done" using a small set of "common jobs", and if yes, characterize these jobs. We use k-means, a well-known data clustering algorithm, to extract such information. We input each job as a data point to k-means, trust the algorithm to find natural clusters, and expect that jobs in a cluster can be described as belonging to a single equivalence class and thus be represented by a single "common job".

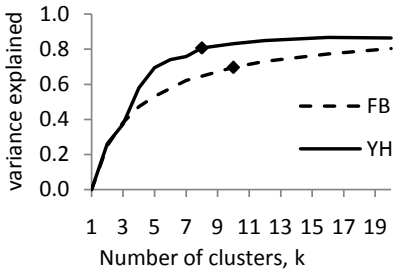

Figure 6: Cluster quality - residual sum of squared distances as the number of clusters increase. The marker indicates the cluster number used for labeling.

Given data points in an n-dimensional space, k-means randomly picks k points as initial cluster centers, assigns data points to their nearest cluster centers, and recomputes new cluster centers via arithmetic means across points in the cluster. K-means iterates the assignment-recompute process until the cluster centers become stationary [1].

The data points are 6-dimensional. Each dimension presents a descriptive characteristic in the original trace - a job's input, shuffle, output data sizes in bytes, its running time in seconds, and its map and reduce time in task-seconds (i.e., 2 parallel map tasks of 10 seconds each is 20 task-seconds). We use Euclidean distance, and linearly normalize all data dimensions to a range between 0 and 1. For each k, we take the best result from 100 random initializations of cluster centers. We pick 100 because we want enough random initializations to cover all 6 data dimensions. Because $100 > 2^6$, each of the 6 data dimensions would have been initialized with at least one "high" and one "low" value. We could have used even more random initializations, but for our data sets, we see only marginal improvements.

We increment k until the cluster quality shows diminishing return. We use the "*variance explained*" clustering quality metric, computed by the total variance in all data points minus the residual within-cluster variance. This metric is useful as it ranges between 0 and 1 when expressed as a fraction of total variance, with 1 being the ideal value.

Figure 6 shows the fraction of variance explained as we increment the number of clusters. Even at small k, we start seeing diminishing returns. We believe a good place to stop is $k = 10$ for the FB trace, and $k = 8$ for the YH trace, indicated by the markers in Figure 6. At these points, the clustering structure explains 70% (FB) and 80% (YH) of the total variance, suggesting that a small set of "common jobs" can indeed cover a large range of behavior.

We can identify the characteristics of these common jobs by looking at the numerical values of the cluster centers. Table 2 shows the cluster size and our manaully applied labels. We defer to Appendix A a more detailed table of the numerical cluster center values and why we selected the cluster labels. We see from Table 2 that both traces have a large cluster of small jobs, and several small clusters of various large jobs. Small jobs dominate the total number of jobs, while large jobs dominate all other dimensions. Thus,

Table 1: Cluster sizes and labels for FB (top) and YH (below). See Appendix A for the numeric values of the cluster centers, and an explanation of how we assigned the cluster labels.

# Jobs	Label
1081918	Small jobs
37038	Load data, fast
2070	Load data, slow
602	Load data, large
180	Load data, huge
6035	Aggregate, fast
379	Aggregate and expand
159	Expand and aggregate
793	Data transform
19	Data summary
21981	Small jobs
838	Aggregate, fast
91	Expand and aggregate
7	Transform and expand
35	Data summary
5	Data summary, large
1303	Data transform
2	Data transform, large

for performance metrics that place equal weights on all jobs, the small jobs should be an optimization priority. However, large jobs should be the priority for performance metrics that weigh each job according to its "size" either in data, running time, or map and reduce task time.

We also see that the FB and YH workloads contain different job types. Small jobs are common. However, the FB trace contains many data loading jobs, characterized by large output data size >> input data size, with minimal shuffle data. The YH trace does not contain this job type. Both traces contain some mixture of jobs performing data aggregation (input > output), expansion (input < output), transformation (input $\approx$ output), and summary (input >> output), with each job type in varying proportions.

Implications for workload model: A small group of "common jobs" can cover a majority of variance in job characteristics within a workload. A good workload model must capture the characteristics and proportions of these jobs. We can run a data clustering algorithm to identify the empirical proportions and multi-dimensional characteristics of each job type in a particular workload,

2.4 An empirical workload model

Combining the insights from previous subsections, we believe the following is a good workload model. The model consists of cluster traces that contain a list of jobs, their submission times, a description of their input data set, and a description of the map and reduce functions of each job. This model has several noteworthy properties.

Completely empirical model: The cluster trace *is* the model. This is a very simplistic approach. It avoids the difficulties of fitting different analytical distributions for data characteristic of different workloads (Section 2.1), or parameterizing analytical job arrival models with empirically observed arrival rates (Section 2.2). If the traces cover diurnal cycles (Section 2.2), then the model automatically captures

the diurnal patterns. The model also caputures the correct mix of job types, and for each job type, the input data properties and properties of the map and reduce functions (Section 2.3). The model relies on good monitoring capabilities to obtain traces of representative behavior, and workload synthesis tools that can operate on purely empirical models. We assume the former, and describe the latter in the next section of the paper.

Can utilize partial information: Even if we have access to production data and code, it would be difficult to get accurate "descriptions of input data set" and "descriptions of map and reduce functions", because both could continuously change. Thus, we must make do with lists of jobs with their corresponding job type, proxy data set, and proxy map/reduce functions for each job type. Sometimes, we have even less information, and our monitoring system records only the data size at the input, shuffle, and output stages. We are then compelled to use generic test data, and proxy map/reduce functions that preserve the data ratios but perform no other computation. Since our model is completely empirical, having partial information only means that less information is passed into the workload synthesis and replay stages. We show later in the paper that using proxy data sets and map/reduce functions can alter performance behavior considerably, but a careful interpretation of the workload replay results would still yield useful insights.

Independent of system characteristics and behavior: Our model specifies the workload without invoking descriptions of system characteristics or behavior. This approach gives us system-independent workloads that allow MapReduce developers to optimize hardware, configurations, schedulers, and other features using different workloads. Our model also leaves as potential optimization targets all behavior characteristics, such as running time, CPU consumption, or data locality. If we include system characteristics in the workload description, we would restrict the workload to only systems with identical characteristics. We also believe that system behavior should not be a part of the workload description at all - running time, CPU consumption, data locality etc. would change from system to system. A workload model that describes system behavior would need to be recalibrated upon any change in the input data, map/reduce function code, or the underlying hardware/software system.

This empirical model of traces of jobs, submission times, proxy data set, and proxy map/reduce functions forms the input to our synthetic workload generator, described next.

3. WORKLOAD SYNTHESIS AND REPLAY

The goal of this section is to present our synthesis and replay pipeline and identify what can be reproduced when we replay a synthetic workload.

We will first provide a brief overview of our goals for modeling, synthesizing and replaying workload (Section 3.1). Next we describe the synthesis and replay pipeline (Section 3.2). We use this pipeline to create synthetic equivalents of a known workload (the Gridmix2 microbenchmark), and replay the synthetic workload in different environments. We explore how performance changes as we selectively vary workload dimensions, such as data size, proxy data sets, and proxy map/reduce functions. We also vary system parameters such as hardware, cluster size, and MapReduce versions. These experiments provide an understanding of what behavior can be reproduced when using synthetic workloads instead of real workload, how much we gain by having map and reduce source code and/or original data, and how to translate observed behavior on one system configuration/workload to other configurations/workloads. (Section 3.3).

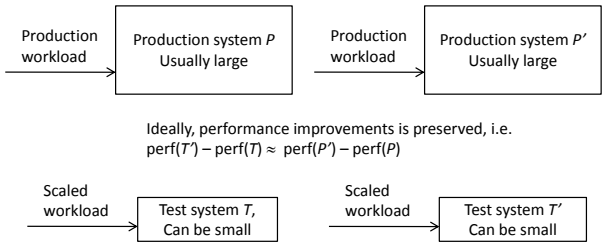

Figure 7: Workload model and replay goals - to drive test systems with a production workload, and ideally have performance improvements across test systems translate back to corresponding production systems.

3.1 Goals of workload modeling

The goal of modeling, synthesizing, and replaying workload is to quantify how changes to the production environment or workload affect performance. Figure 7 summarizes a common situation. We have a production system P, which runs some workload, and we need to know how performance changes in a modified production system P' which will run the same workload. However, building P' directly represents an expensive commitment with no guaranteed performance improvements. We build test systems T and T', with the same modifications, drive them with the same workload, and observe any performance differences. Ideally, the workload and the test system are constructed in a way such that the performance differences between T and T' will be preserved between P and P'.

3.2 Model-synthesis-replay pipeline

We use the model-synthesis-replay pipeline in Figure 1. As described in Section 2, we use a completely empirical model, in which the system traces *are* the model. The traces should contain at least a list of jobs, their submit times, some description of the data set, and some description of the map and reduce functions.

We input these traces and a desirable synthetic workload length L into a synthetic workload generator. We take samples of the trace, each sample covering a continuous time window of W. The contatenation of $L/W = N$ samples creates our synthetic workload. The synthetic workload is also a list of jobs, their submit times, some description of the data set, and some description of the map and reduce functions. We emphasize that this list is generated a-priori and not while the synthetic workload runs in real time. We replay this workload by submiting the list of jobs at their corresponding submit times, operating on their corresponding data set, performing the corresponding map and reduce functions. It is feasible to read in the job list and submit jobs in real time because the job launch rate is at most one job every few seconds or so (Figure 4). We measure sys-

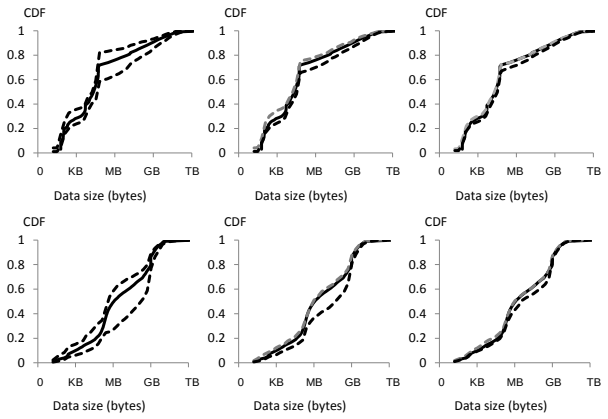

Figure 8: Distributions of FB input data size (top) and output data size (below), showing max/min sample CDFs (dashed lines) converging to the total aggregate CDF (solid line) in 6×4-hrs samples (left), 24×1-hr samples (middle), and 96×15-minutes samples (right).

tem performance during the synthetic workload replay, then decide whether the deployment/workload changes are appropriate for the production cluster. We elaborate several subtle considerations below.

The choice of L and W involve some tradeoffs. We want a short L to faciliate a rapid design-measurement-redesign cycle. We also want a large W to capture job sequences and bursts. However, when L is fixed, large W means small N and few samples, and consequently less representative behavior in the synthetic workload. Fortunately, the deviation from "representative behavior" is statistically bounded, as we demonstrate below.

We use $F(x)$ to denote the statistical distribution of some job characteristic, e.g. one of the CDFs in Figure 2. $F(x)$ is the "representative" system behavior. We use $F_N(x)$ to denote the same distribution computed over our synthetic workload of N samples, i.e., the "sampled" system behavior. As N increases towards infinity, $F_N(x)$ converges to $F(x)$. However, for finite N, the variance in $F_N(x)$ is

$$F_N(x) = \frac{F(x)(1 - F(x))}{N}$$

As N increases, the confidence interval around $F_N(x)$ converges at $O(1/\sqrt{(}N))$ [18]. Thus, if we increase N four times by decreasing W four times, we will halve the deviation of sampled behavior $F_N(x)$ from representative behavior $F(x)$. We empirically verified these bounds. Figure 8 shows 10 repeated samples of $F_N(x)$ for consecutive four-fold increases in N. The maximum and minimum $F_N(x)$ converges to $F(x)$ very quickly. For brevity, Figure 8 shows only the convergence for two CDFs. However, we verified the same behavior for other dimensions as well.

In reality, it may be logistically infeasible to reproduce all the map and reduce functions, as well as the input data set, even for organizations that operate their own production clusters. Ideally, there would be enough information to identify common data types and common jobs. Based on such information, we can construct proxy data sets and proxy map and reduce functions for each job type, e.g. a load function, an aggregate function, etc. running on natural text, image files, web indices, etc. Often, we lack even that level of visibility if we only use Hadoop's built-in tracing tools. We can resort to using random bytes as input data, and using data-ratio preserving pipes as proxy map and reduce functions. We show later that for IO intensive jobs, data-ratio preserving proxy functions approximate task running times reasonably well.

3.3 Replay known workload

We describe several experiments to quantify how our workload modeling, synthesis and replay methodology reproduces behavior across changes to the underlying MapReduce system or the workload. We replay a known workload across changes in workload data size, cluster hardware, MapReduce version, and proxy functions for map and reduce. Our "known workload" is Gridmix 2, a microbenchmark included with recent Hadoop distributions. See Appendix B for a list of jobs in Gridmix 2. For our following discussion, we observe overall patterns across different jobs.

Reference system: For a comparison baseline, we run Gridmix 2 on a local 20-machine cluster testbed, *Local-cluster-A*. The machines are 3GHz dual-core Intel Xeons, with 2GB memory and 1Gbps network connection. Gridmix 2 repeats different jobs for a different number of times. We take the arithmetic average of the repeats. We run Gridmix 2 using Hadoop 0.20.2 under default configurations.

Cluster and data scaling: We study the impact of halving the size of the cluster and the data. This experiment preserves the data intensity per machine, essentially creating the same "workload" at a different scale. We expect that as we scale the cluster and data sizes, the map and reduce times scale by the same factor. Figure 9 compares the replay and reference map and reduce times in task-seconds (2 tasks of 10 seconds each is 20 task seconds). The horizontal axis represents map/reduce time on the reference system, and the vertical axis represents the corresponding map/reduce time for the replay. Different markers represent different job types in Gridmix 2. The top diagonal represents the 1-to-1 diagonal. The lower diagonal represents the 2-to-1 diagonal, i.e., replay takes half as long. We expect the markers to lie on the lower, 2-to-1 diagonal. This expectation holds true for map times. However, reduce times mostly fall between the 1-to-1 diagonal and 2-to-1 diagonal. Reduce time did not scale down with data size because Gridmix 2 configures reduce task counts for a larger, 2TB input data set. This results in a tiny amount of data assigned per reduce task, with reduce time dominated by launch and completion overhead. **Key observation: Under good configurations, map and reduce times should scale with data size, i.e., half the data requires half the work**.

Different hardware: We run the half-data-size Gridmix 2 workload in a second local cluster (*Local-cluster-B*). These machines are 2.2GHz quad-core Opterons, with 4GB memory and 1Gbps network. We expect that the map and reduce times will scale with data size with no distortion. Figure 10 shows the replay results. All markers fall on the 2-to-1 diagonal - map and reduce times scale with data size. **Key observation: If there are no hardware bottlenecks,**

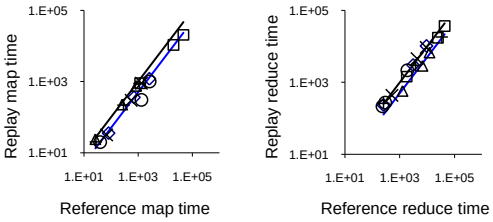

Figure 9: Map time (left) and reduce time (right) comparisons for replaying Gridmix 2 with halved cluster and data size. Different markers represent different job types in Gridmix 2.

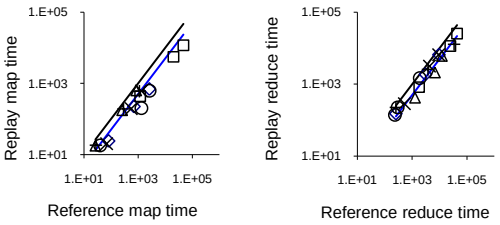

Figure 10: Map time (left) and reduce time (right) comparisons for replaying Gridmix 2 with halved cluster and data size and on different hardware. Different markers represent different job types in Gridmix 2.

map and reduce times should scale with data size.

Different Hadoop version: We now change the Hadoop version. We run the half-data-size Gridmix 2 on Local-cluster-B with Hadoop 0.18.2, an earlier Hadoop version. Hadoop 0.20.2 in the reference system has many improvements, bug fixes, and new features. Figure 11 shows the replay results. Interestingly, for map times, all markers are below the 2-to-1 diagonal. Changes from Hadoop 0.18.2 to Hadoop 0.20.2 actually *increased* map times for jobs in Gridmix 2 under default configurations. One possible reason is that the new features create additional overhead. **Key observation: New Hadoop versions may not improve performance for all workloads and all deployments.**

Proxy map and reduce functions: Lastly, we quantify the effects of using "proxy" map and reduce functions that use random bytes for input, preserve data ratios, but performing no additional computation. These proxy functions are necessary when we do not have enough information from workload traces to develop representative functions for each common job type. Gridmix 2 uses similar load generators. We wrote our own data sampler and pipes to see the effects of running a different piece of code for performing similar computations on the same data set. These experiments use Hadoop 0.18.2 on Local-cluster-B. We expect that the general performance trend/ordering with respect to per-job map and reduce times is preserved. Figure 12 shows the replay results. The markers are no longer on a tight diagonal line, but in a loose, roughly diagonal region. For some job types, the markers for three different job sizes do not even form an diagonally increasing line. The statistical correspondance is much more loose, but still present. **Key observation: Even without access to the production code and data, we can replay some workloads using**

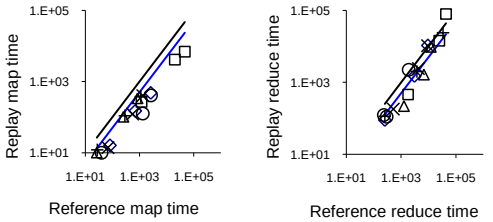

Figure 11: Map time (left) and reduce time (right) comparisons for replaying half-data-size Gridmix 2 with a new Hadoop version. Different markers represent different job types in Gridmix 2.

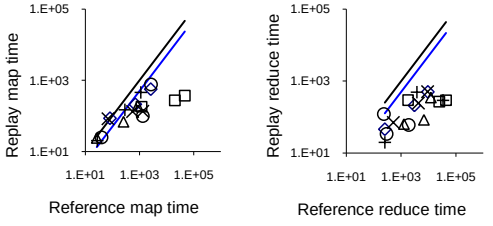

Figure 12: Map time (left) and reduce time (right) comparisons for replaying half-data-size Gridmix 2 using proxy map and reduce functions that only pipe the data according to specified data ratios. Different markers represent different job types in Gridmix 2.

proxy map and reduce functions and perserve the overall performance ordering, i.e. longer jobs still take longer overall.

3.4 Summary of replay effectiveness

Our replay experiments in Section 3.3 represent a systematic way to explore MapReduce performance across various dimensions of workload and system characteristics. If we take the reference system to be a "production system", then we see incrementally larger performance differences as we abstract away data size and cluster scale, hardware, software, and "production" code. However, even when we use proxy map and reduce functions, the performance ordering between jobs is still roughly preserved. The underlying assumption is that the various systems are all replaying the same Gridmix 2 "workload".

We believe that workload replay should *not* try to reproduce the exact resource utilization, running time, and other system behavior. Section 3.3 demonstrated that exact replication is difficult when we change system characteristics and use proxy map/reduce functions and data sets. Moreover, many proposed MapReduce improvements seek to optimize system behavior. Fortunately, the methodology in Figure 7 does not require exact behavior replication. It requires only that the workload replay preserve the ordering of performance metrics computed across all jobs in the workload. Ideally, we would also preserve the approximate magnitude of performance differences. In the next section, we investigate whether both these properties are achievable using a synthetic workload based on the FB trace.

4. WORKLOAD-LEVEL MEASUREMENT

In this section, we apply the model-synthesis-replay methodology to an example use case. The goal is to demonstrate that the methodology leads to surprising insights, and to provide a reference point to extend the method to other workloads and use cases. We describe the example use case, which involves operators of large scale EC2 MapReduce clusters quantifying performance improvements for a "Facebook-like" workload (Section 4.1). We describe the workload, the systems measured, and the performance metrics. We then present the performance measurements (Section 4.2). We show that system behavior and performance ordering translate across test and production clusters, i.e., the system setting that performs better on the test cluster also performs better on the production cluster. Two surprises are 1. job failure rate is an important metric, 2. MapReduce schedulers have a bigger than expected impact. Finally, we describe how MapReduce developers can extend the methodology to their own workloads and use cases (Section 4.3). We give concrete examples of how to answer "what-if" questions regarding the choice of hardware and configurations, or changes in the workload scale and composition.

4.1 Use case description

We are operators of a large scale EC2 cluster running a "Facebook-like" MapReduce workload. We are running Hadoop 0.18.2 with default configurations on a cluster of 200 m1.large instances. We want to find out what would be the performance improvement if we upgrade to Hadoop 0.20.2 with tuned, non-default configurations. We do not want to do a fork-lift upgrade of the large cluster until we are sure that there are significant performance improvements. We have financial resources to do short-term performance testing on a small scale cluster, but not enough resources to do testing on a mirror large scale cluster, nor conduct long-term measurements and comparisons.

This use case captures common performance testing challenges for MapReduce cluster operators. Successfully addressing the use case requires exercising all aspects of our MapReduce workload performance methodology. Below are some more details of the use case.

4.1.1 "Facebook-like" workload

Our workload model is the FB trace from Section 2. This trace describes only the data sizes at each input, shuffle, and output stages. We do not have access to the original map/reduce functions or the production data set at Facebook. Hence our "Facebook-like" workload operates on input data of random bytes, using data ratio preserving proxy map/reduce functions. We emphasize this synthetic workload is not a Facebook workload - essential information about map/reduce functions and the input production data set is missing. It is a "Facebook-like" workload that has the same job submission sequences, arrival intensities, data size, and data patterns as the original Facebook workload. In our use case, our large scale EC2 cluster runs the "Facebook-like" workload.

To make sure the workload fits on a small scale test cluster, we scale the data size by the scaling factor of cluster size. As explained in Section 3.2, this scaling preserves the workload compute intensity - a fraction of the workers (cluster size) is doing the corresponding fraction of work (data size). Similarly, the continuous time-window sampling method allows us to capture representative workload properties with short synthetic workloads and statistically bounded deviations. We want rapid experimentation. Thus, we produce a day-long synthetic workload using hour-long continuous samples of the workload trace.

4.1.2 Production and test systems

We follow the performance comparison illustrated in Figure 7. The production system P runs Hadoop 0.18.2 with default configurations on 200 m1.large EC2 instances. The test system T has identical characteristics, except the cluster size is decreased to 10 instances. Each EC2 instance has 7.5GB memory, equivalent CPU capacity of 4×1.0-1.2GHz Opteron processors, and "High" IO performance.

We want to know the performance of production system P' that runs Hadoop 0.20.2, also on 200 m1.large EC2 instances. However, the Hadoop configurations would be tuned. Appendix C lists the tuned configuration values, and the reasons for changing them from the default. Our small scale test cluster T' has identical characteristics, except the cluster size is decreased to 10 instances.

We know from Hadoop change logs that there have been many improvements and new features going from Hadoop 0.18.2 to Hadoop 0.20.2. Our measurement results show that the most important change for our "Facebook-like" workload is the switch from the FIFO scheduler that assigns all available cluster resources to jobs in FIFO order, to the fair scheduler that seeks to give each active job a concurrent fair share of the cluster resources.

4.1.3 Performance metrics

We use three performance metrics. The first metric is an efficiency metric - the map times and reduce times summed across all jobs. The less task-time the same workload takes, the more work can fit onto the same cluster.

The second metric is a latency metric - the per-job running time, computed from job submission to job finish. This metric represents the wait time for MapReduce results. The latency metric is more complicated than the efficiency metric. We may be willing to sacrifice an increase in average latency for a decrease in worst-case latency. These trade-offs are especially relevant since the "Facebook-like" workload has a few very large jobs mixed in with many small jobs. A latency increase from, say, 20 to 30 seconds may be tolerable in exchange for latency decrease elsewhere from 3 hours to 2 hours.

It turns out that a third metric takes complete precedence - the percentage of failed jobs. We had assumed that all system settings can complete all jobs. However, the FIFO scheduler in Hadoop 0.18.2 led to many small jobs after large jobs failing, while the fair scheduler in Hadoop 0.20.2 allowed the same jobs to complete. While we were not surprised that the fair scheduler was superior, we were very surprised that the job failure rate metric was so essential.

4.2 Measurement results

We can visually verify that the observed performance differences on test clusters T and T' translate to production clus-

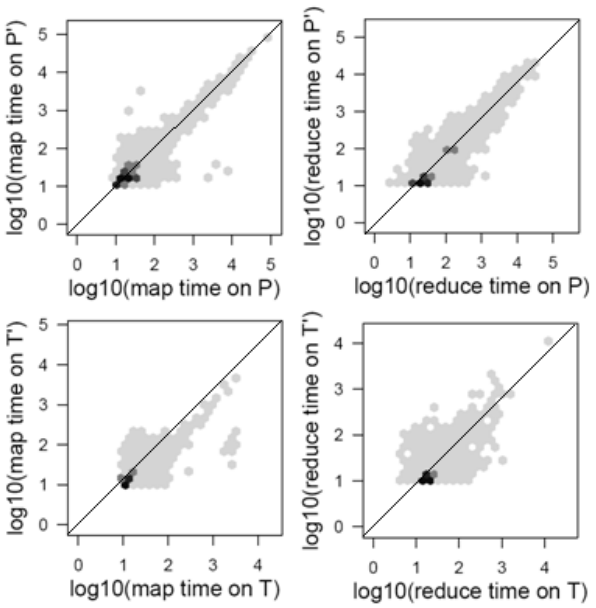

Figure 13: Map time and reduce time comparisons between test clusters T vs. T' (bottom), and between production clusters P vs. P' (top). In each column, the similar shapes and locations of the darkest bins show that the performance differences between test clusters translates to production clusters. See the beginning of Section 4.2 for detailed discussion.

ters P and P'. Figure 13 shows the map and reduce time comparisons between test clusters T vs. T', and production clusters P vs. P'. Each graph compares the map/reduce time in Hadoop 0.18.2 (horizontal axis) vs. Hadoop 0.20.2 (vertical axis). The graphs show log10 values in hexgonal bins, with darker colors meaning more data points in the bin. We also include the reference 1-to-1 diagonal. Dark bins below the diagonal indicates a performance improvement for many jobs going from Hadoop 0.18.2 to 0.20.2. If performance differences translate from test to production clusters, then graphs in each column should show similar shape of distribution for all bins, with roughly matching locations for the densest bins. This is indeed the case. In the left column comparing map times, both the top and bottom graphs show all bins group around the diagonal, with the densest bins also located around the diagonal. In the right column comparing reduce times, both top and bottom graphs also show all bins group around the diagonal, with the densest bins located below the diagonal.

Figure 13 gives qualitative indication that observed behavior translates. We outline below a more rigorous, quantitative comparison of the job failure, efficiency, and latency metrics.

4.2.1 Job failure and efficiency

The job failure comparison was striking. On test cluster T running Hadoop 0.18.2, 5.5% of jobs did not complete, while only 0.1% of jobs failed on test cluster T' running Hadoop 0.20.2. This ordering is preserved on production clusters, with 8.4% of jobs failing on production cluster P and 0.7% of jobs failing on production cluster P'. While we were not surprised that the ordering of failure rates is preserved, we find it striking that using Hadoop 0.20.2 cuts the failure rate by an order of magnitude. More detailed examination allowed us to identify a failure mode explained by the difference between the FIFO and fair scheduler.

The vast difference in job failure rates complicates a more rigorous comparison of efficiency and latency. First, we can meaningfully compare only those jobs that successfully completed in both T and T', or P and P'. Second, more subtly, job failures in fact lighten the load on the cluster. When a job fails, the remainder of the work for that job is removed from the workload and no longer loads the cluster. Thus, successful jobs running immediately after job failures would see lighter than expected cluster load. However, if the all jobs had run to completion, then all jobs would see the same cluster load. The precise performance ordering would depend on the balance of better schedulers that improve efficiency and latency, bad schedulers that "improve" efficiency and latency by "removing" jobs from the workload through job failures, and other differences between the two system settings under comparison.

We include here the efficiency comparisons for jobs that are successful in both Hadoop 0.18.2 and 0.20.2, with an emphasis that the comparison is less reliable for the reasons listed above. Upgrading from Hadoop 0.18.2 on T to 0.20.2 on T' sees an average of 24% improvement in map times and 22% improvement in reduce times. The corresponding upgrade from P to P' sees an average of 3% improvement in map time and 31% improvement in reduce time.

We take a more detailed look at the effect of the fair scheduler below.

4.2.2 Impact of schedulers

Briefly, MapReduce task schedulers work as follows. Each job breaks down into many map and reduce tasks, operating on a partition of the input, shuffle, and output data. These tasks execute in parallel on different worker machines on the cluster. Each machine has a fixed number of task slots, by default 2 map tasks and 2 reduce tasks per machine. The task scheduler sits on the Hadoop master, which receives job submission requests and coordinates the worker machines. The FIFO scheduler assigns all available task slots to jobs in FIFO order, while the fair schedule seeks to give each active job a concurrent fair share of the task slots. The biggest performance difference occurs when the job stream contains many small jobs following a big job. Under FIFO, the big job would take up all the task slots, with the small jobs enqueued until the big job completes. Under the fair scheduler, the big and small jobs would share the task slots equally, with the big jobs taking longer, but small jobs being able to run immediately.

Figure 14 shows three illustrative job sequences and their run times under FIFO and fair schedulers.

In the top graph, several bursts of large jobs cause many jobs to fail for the FIFO scheduler, while the fair scheduler operates unaffected. Under FIFO, subsequent arrivals of small jobs steadily lengthens the job queue. There are several failure modes, and we have not yet pin-pointed the cause of every one. In one common failure mode, jobs fail

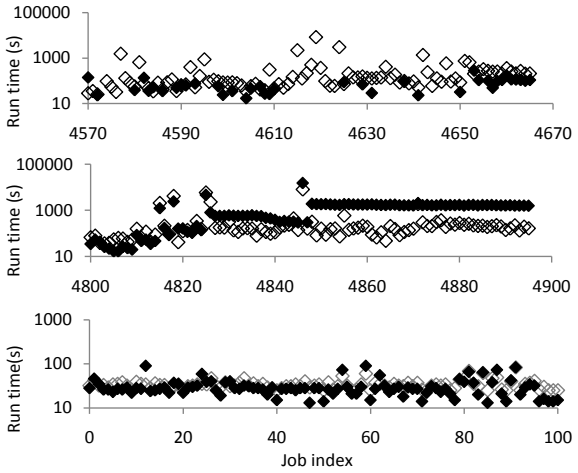

Figure 14: Three illustrative job sequences for comparing the FIFO scheduler (solid markers) and the fair scheduler (hollow markers), showing job failures in the FIFO scheduler (top), unnecessarily long latency in the FIFO scheduler (middle), and slight latency increase for small jobs in Hadoop 0.20.2 (bottom).

because the the entire cluster runs out of disk space. The disk holds the working set of shuffle data of all active jobs. Having a large job queue can increase this working set considerably, with earlier jobs in the reduce phase operating on old shuffle data, but subsequent jobs writing additional shuffle data using free map slots. Full disks across the cluster cause jobs to fail despite re-submission and recovery mechanisms, until enough jobs have failed to cause intermediate data to be cleared and disk space to be freed.

Using very large disks can avoid this failure (the disks in the experiment are already 400GB). However, switching to the fair scheduler considerably lowers the disk space requirements, since all jobs have an equal chance to finish, allowing the working set of completed jobs to be reclaimed. However, even for the fair scheduler, we still observe the failure mode, just much more rarely. This illustates that running the synthetic workload can test the correct system sizing under realistic job sequences and data intensities, as we have identified a disk space limitation here.

The top graph also shows that successful jobs see lighter than expected cluster load when submitted immediately after strings of job failures. The running times for Jobs 4650 onwards all show that jobs using the FIFO scheduler completed faster. The reason is that preceding job failures removed otherwise still active cluster loads! When the failure rates differ so greatly, we believe the failure rate metric should take precedence over efficiency and latency metrics.

The middle graph shows the precise job arrival pattern that the fair scheduler was designed to optimize. Several very big jobs arrive in succession (the high markers around Job 4820 and another just beyond Job 4845). Each arrival brings a large jump in the FIFO scheduler finishing time of subsequent jobs. This is again due to FIFO head-of-queue blocking. New jobs continue to lengthen the queue before old jobs can drain. Once the head-of-queue large job completes, all subsequent small jobs complete in rapid succession, leading to the horizontal row of markers. The fair scheduler, in contrast, shows small jobs with unaffected running times, sometimes orders of magnitude faster than their FIFO counterpart. Such improvements is in agreement with the best-case improvement reported in the original fair scheduler paper [19], but far higher than the average improvement reported there.

The lower graph shows the finishing time of small jobs during times of low load (note the different vertical axis). In this context, Hadoop 0.20.2 is slower than Hadoop 0.18.2, unsurprising given the many added features since 0.18.2. The fair scheduler brings little benefit in these settings. However, in this workload, low load periods occur more frequently than high load periods, meaning that the vast improvements during high load may averaged out into performance penalties.

Based on these observations, we decide for the use case that we should upgrade to Hadoop 0.20.2. We further recommend that the fair scheduler should be the default scheduler for workloads with similar patterns of small jobs mixed with large jobs. The order-of-magnitude latency benefit for all jobs during load peaks far outweights latency increases of a few tens of seconds for small jobs during common periods of light load.

The original fair scheduler paper [19] could not perform this analysis because the micro-benchmarks used there do not sufficiently capture job arrival sequences. In contrast, our continuous time-window trace sampling method does.

4.3 Extension to other use cases

Given our observations on a "Facebook-like" workload, it is not surprising that the fair scheduler was initially developed at Facebook [19]. Facebook and other MapReduce operators can employ the same methodology for other workloads. They can generate synthetic test workloads based on their own production traces, collected using built-in Hadoop tracing tools. If necessary, MapReduce operators can perform k-means clustering to identify common jobs. Access to production code and production data set will facilitate better proxy map/reduce functions and test data sets. MapReduce operators can run these synthetic workloads on small scale test systems, quickly identify any system sizing issues, while doing performance comparisons under system changes and optimizations. If the test systems show a significant performance improvement, it may not guarantee the same performance improvement on the production system, but it will at least provide good reason to make capital and time investment in testing with larger scale systems.

Furthermore, a straight forward extension of the methodology would be to keep the system fixed, but scale up the workload by increasing the job intensity or data size for different job types within the workload, or add new job types to the workload. Such performance measurements allow MapReduce operators to rapidly explore "what if" scenarios under projected workload growth, or consolidation of multiple workloads on a single cluster. Such capabilities provide invaluable assistance for cluster capacity planning and similar activities.

Many MapReduce operators already do similar performance measurements. Our contribution is the formulation of the empirical model components, the systematic way to identify common jobs and data types, the workload generation methods that create scalable, system-independent workloads, a workload-level performance measurement and interpretation method, and the broader conceptual framework that facilitates performance comparisons across different MapReduce deployments and use cases.

5. RELATED WORK

5.1 MapReduce improvements

We have already done extensive comparisons with the Hadoop fair scheduler [19]. We discussed in detail the significance of using a workload-oriented measurement approach. In our opinion, the evaluation method in the fair scheduler work represents the closest approximation to a workload-level performance evaluation. The authors did make an effort to identify common jobs within a production trace, and generated job sequences by sampling empirical CDFs of job inter-arrival times and per-job task numbers. However, the authors used memoryless Poisson sampling of inter-arrival times. While correctly capturing the overall distribution, memoryless sampling fails to capture realistic workload bursts (Figure 5), nor realistic sequences of small jobs behind big jobs (Figure 14), the exact job arrival pattern that benefits most from the fair scheduler. Another major shortcoming is using several microbenchmarks as proxy map/reduce functions that may not match the data ratios in the production trace. Further, there is no method to scale down the workload, implying costly, large scale development and testing experiments on EC2.

In Mantri [4], the authors evaluated their mechanisms on a production Bing cluster for several months. The work brings immediate and demonstratable improvement on production systems. However, as explained in the paper introduction, we also need small scale, rapid experiments isolated from the production cluster, requiring smaller capital and time investment, but with methodology robust enough that improvements on the test cluster is likely to translate to the production cluster. Likewise, we need a general workload description framework so that we can understand which subset of the observed improvements are limited to the particular production cluster and workload, and which improvements are more generalizable.

Recent work on energy proportional MapReduce clusters proposed an "all-in strategy" that shuts down the cluster when there are no active jobs, but leave the cluster fully active otherwise [9]. Figure 4 suggests that energy proportionality is indeed *highly* desirable, since the peak-to-average ratios are quite high in all workload dimensions. However, a simple calculation from average running times and average job inter-arrival times suggest that the cluster will always have several active jobs. In other words, the "all-in strategy" would almost never shut down the cluster. The workload insights suggest the need for more sophisticated mechanisms to achieve cluster energy proprotionality.

There is also work on progress indicators for MapReduce jobs [11]. The key idea is to estimate progress based on subsets of key-value pairs that have already gone through the MapReduce pipeline, either from a partially completed, ongoing execution, or from already completed debug runs on data sub-samples. This approach is effective for a workload dominated by large jobs, a use case specifically targeted by the authors. However, on workloads dominated by small jobs, such as the FB and YH workloads, the small jobs may be already complete by the time we have enough information for a progress estimate. Thus, we either declare that small jobs do not need progress indicators, or we need progress estimate mechanisms targeted at small jobs.

Other works on MapReduce that use only a microbenchmarks suite (e.g., [2, 15]) can benefit similarly from workload-level insights. Existing microbenchmarks such as Gridmix, PigMix [14], and the Hive Benchmark [8] contain only a handful of jobs. Our workload comparison earlier shows that unless parameterized with the right empirical data size and job proportions, a handful of jobs captures only a subset of workload behavior.

5.2 General workload characterization

Our view on the distinction between workload and behavior characteristics contrasts with the position in a recent work that look at Google cluster workloads [10]. There, the authors specifically want to build workload descriptions that include behavior characteristics such as CPU and memory. The approach is fine given that the authors have access to production data and code. However, without such knowledge, outside readers would have a hard time replicating similar workloads - "high CPU" or "low memory" means very little once the workload runs on different hardware, software, data set, and functionally equivalent code. Even within Google, behavior-based workload descriptions need to be re-calibrated every time the system changes.

Another recent workload characterization effort is the Yahoo Cloud Serving Benchmark [5]. The focus there is characterizing the activity of database-like systems at the read/write level. The main differences with our work include the choice of several analytical distributions to describe data record selection - the user still needs traces of his own workload to decide which (or none) of distribution is appropriate. The benchmark also has read/write ratios that need to be similarly configured per workload. Additionally, the benchmark runs the workloads with the same intensity over time, thus capturing average behavior but not the sequence of busy and idle periods.

Our workload modeling approach has much in common with the approaches in Internet measurement and simulation literature. We focus on workload characteristics that are independent of system characteristics and system behavior. In the Internet simulation literature, this approach gives "source models" that describe the end points and data size of any communication patterns. The contrasting approach is "packet models" that describe how the data is actually sent. The debate settled firmly in favor of source models [13], because packet models capture system behavior that need to be recalibrated across different systems, the same reason that we exclude system and behavior characteristics from our models. Additionally, the earliest traffic models are empirical, like our MapReduce models here, with analytical models being gradually developed as researchers gain more

insight into different Internet workloads [12, 6].

Dynamic web content benchmarks also focus on workload characteristics independent of system characteristics and behavior [3, 17, 16]. There, the benchmarks describe web workloads in terms of user sessions, web objects, request and transaction patterns, database size, attributes, and relationships, and the like. More importantly, these benchmarks specify a suite of common application workloads, including banking, e-commerce, auction site, bulletin board, support, and others. A MapReduce benchmark comparable to TPC-W and SPECweb in scope and coverage would be a great catalyst for innovations.

6. SUMMARY AND CONCLUSIONS

The contributions of our work are multi-fold - (1) an empirical model for characterizing MapReduce workloads, (2) algorithms to generate short, scalable, and representative synthetic workloads, and (3) replay mechanisms that enable performance comparisons across various system and workload changes. The key take-aways for our readers are summarized below:

Why is workload-level measurement methodology necessary? MapReduce workload can differ greatly in many dimensions (Section 2). Without per-workload performance measurements, we cannot confidently quantify the performance impacts of proposed system changes (Section 4).

What does such a methodology involve? Tracing a production workload using several characteristics (Section 2.4), generating short synthetic workloads that captures representative behavior (Section 3.2), and replaying the workload on small scale test clusters before trying the same workload at large scale (Section 4).

How can MapReduce developers implement and apply this methodology for a particular workload? Trace a production workload and record at least job submission times, and per-job data size for input, shuffle, and output. If possible, also record job names, so that developers can identify "common jobs" in the workload (Section 2.3), and construct proxy data sets and map/reduce functions for each job type. Synthetic workload replay should replicate, on the test cluster, as many characteristics of the production cluster as appropriate or feasible, except for the system/workload change being tested and the behavior characteristics being optimized. Improved performance on the test cluster is likely to translate to the production cluster, even though the exact magnitude of the improvement may not translate. The workload-level methodology provides information to justify capital and time investment in testing on a large scale cluster (Section 4).

As MapReduce applications and deployment environments continue to diversify, workload and system specific performance improvements become more difficult to generalize. Our contributions with regard to trace samples, scaled workloads, proxy functions and test data sets are a step towards a TPC-W or SPECweb style MapReduce workload suite, and can serve as both a generic and a application-specific MapReduce benchmark. Furthermore, our methology allows operators to publish MapReduce traces while protecting proprietary information about the specific map and reduce computations performed. We hope our analysis of two production workloads inspires other production environments to share traces and insights about their systems, including and beyond MapReduce.

7. REFERENCES

[1] E. Alpaydin. *Introduction to Machine Learning*. MIT Press, Cambridge, Massachusetts, 2004.

[2] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys 2010*.

[3] C. Amza, A. Chanda, A. Cox, S. Elnikety, R. Gil, K. Rajamani, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic web site benchmarks. In *IEEE International Workshop on Workload Characterization*, 2002.

[4] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI 2010*.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC 2010*.

[6] P. B. Danzig, S. Jamin, R. Cáceres, D. J. Mitzel, and D. Estrin. An empirical workload model for driving wide-area tcp/ip network simulations. *Internetworking: Research and Experience*, 3:1–26, 1992.

[7] Hadoop Wiki. Hadoop Power-By Page. http://wiki.apache.org/hadoop/PoweredBy.

[8] Y. Jia and Z. Shao. A Benchmark for Hive, PIG and Hadoop. https://issues.apache.org/jira/browse/hive-396.

[9] W. Lang and J. Patel. Energy management for mapreduce clusters. In *VLDB 2010*.

[10] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *SIGMETRICS Perform. Eval. Rev.*, 37:34–41, March 2010.

[11] K. Morton, M. Balazinska, and D. Grossman. Paratimer: a progress indicator for mapreduce dags. In *SIGMOD 2010*.

[12] V. Paxson. Empirically derived analytic models of wide-area tcp connections. *IEEE/ACM Trans. Netw.*, 2:316–336, August 1994.

[13] V. Paxson and S. Floyd. Why we don't know how to simulate the internet. In *Proceedings of the 29th conference on Winter simulation*, 1997.

[14] Pig Wiki. Pig Mix benchmark. http://wiki.apache.org/pig/PigMix.

[15] T. Sandholm and K. Lai. Mapreduce optimization using regulated dynamic prioritization. In *SIGMETRICS 2009*.

[16] Standard Performance Evaluation Corporation. SPECweb2005. http://www.spec.org/web2005/.

[17] Transaction Processing Council. TPC-W. http://www.tpc.org/tpcw/.

[18] L. Wasserman. *All of Statistics*. Springer, New York, New York, 2004.

[19] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys 2010*.

[20] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI 2008*.

APPENDIX

A. LABELS FOR COMMON JOBS

The numerical values for cluster centers are in Table 2. We represent cluster centers using medians instead of arithmetic means because the data in each dimension is very long-tailed. For such data, the median represents a "center" measurement that is more robust against data outliers. The cluster labels, and reasons for labeling, are as below.

FB trace:

- Small jobs: Small values for all dimensions.
- Load data, fast: Input << output, no shuffle, short duration.
- Load data, slow: Input << output, no shuffle, long duration.
- Load data, large: Small input, no shuffle, very large output.
- Load data, huge: Small input, no shuffle, huge output.
- Aggregate, fast: Input >> shuffle >> output, short duration.
- Aggregate and expand: Input >> shuffle << output.
- Expand and aggregate: Input << shuffle >> output.
- Data transform: Input ≈ shuffle ≈ output.
- Data summary: Huge input, very small output.

YH trace:

- Small jobs: Small values for all dimensions.
- Aggregate, fast: Input >> shuffle, shuffle >> output, short duration.
- Expand and aggregate: Input << shuffle >> output.
- Transform and expand: Input ≈ shuffle << output.
- Data summary: Huge input, very small output.
- Data summary, large: Even bigger input, very small output.
- Data transform: Input ≈ shuffle ≈ output.
- Data transform, large: Input ≈ shuffle ≈ output, big data.

B. GRIDMIX JOBS

Table 3 list the jobs in Gridmix 2 with a brief description and the input-shuffle-output data ratios for each job.

Table 3: Jobs in Gridmix 2, showing input-shuffle-output (I-S-O) ratios.

Job name	I-S-O ratio
Three-stages job, output of one job is input to the next job	1.00-0.10-0.04, then 1.00-1.00-0.77, then 1.00-1.16-1.06
Web data sort with variable sized keys and values.	1.00-1.00-1.00
Reference select, i.e. scan and select from a large data set	1.00-0.002-0.0001
Text sort, using java, pipe, and streaming APIs	1.00-1.00-1.00
Combiner/wordcount. Added to Gridmix2.	Data dependent. Small shuffle/output

C. TUNED CONFIG. FOR HADOOP 0.20.2

We list the non-default configuration values, and why we changed them.

- mapred.jobtracker.taskScheduler: Default FIFO, changed to fair scheduler to do performance comparison between the two schedulers.
- mapred.job.tracker.handler.count: Default 10, changed to 20 to accomodate large scale clusters.
- tasktracker.http.threads: Default 40, changed to 50 to accomodate large scale clusters - more threads responding to more task trackers.
- mapred.child.java.opts: Default -Xmx200m, changed to -Xmx1512m to increase JVM memory from 200MB to 1.5GB.
- dfs.block.size: Default 64MB, changed to 128MB to decrease overhead for jobs operating on big data.
- io.file.buffer.size: Default 4KB, changed to 64KB to hopefully increase IO performance.
- io.sort.factor: Default 10, changed to 15 to allow more parallel shuffles during the implicit sort phase for reduce tasks.
- io.sort.mb: Default 100MB, changed to 200MB to hopefully increase sort performance.

Table 2: Cluster sizes, medians, and labels for FB (top) and YH (below). Data sizes are in KBs, durations are in seconds, and map/reduce times are in task-seconds, i.e. 2 tasks of 10 seconds each is 20 task-seconds.

# Jobs	Input	Shuffle	Output	Duration	Map time	Reduce time	Label
1081918	21	0	871	32	20	0	Small jobs
37038	381	0	1,900,814	1288	6079	0	Load data, fast
2070	10	0	4,217,618	6194	26321	0	Load data, slow
602	405	0	447,303,306	4381	66657	0	Load data, large
180	446	0	1,100,929,790	18362	125662	0	Load data, huge
6035	230,903,926	8,776,400	491,141	906	104338	66760	Aggregate, fast
379	1,916,392,963	502,594	2,590,621	1916	348942	76736	Aggregate and expand
159	417,520,076	2,510,696,562	44,624,096	5024	1076089	974395	Expand and aggregate
793	254,860,808	788,155,641	1,594,722	2190	384562	338050	Data transform
19	7,579,267,317	51,840,780	104	3408	4843452	853911	Data summary
21981	174,249	73,142	6,346	63	412	740	Small jobs
838	568,193,339	75,630,299	3,985,889	2107	270376	589385	Aggregate, fast
91	206,254,306	1,540,406,112	132,942	2426	983998	1425941	Expand and aggregate
7	806,443,777	235,378,858	10,005,568,924	9412	257567	979181	Transform and expand
35	4,986,289,455	77,906,463	775,483	13585	4481926	1663358	Data summary
5	31,112,762,354	937,362,456	475,085	30890	33606055	31884004	Data summary, large
1303	35,804,623	14,970,878	4,049,293	3539	15021	13614	Data transform
2	5,487,309,879	10,347,599,865	2,461,853,150	16687	7729409	8305880	Data transform, large