Towards Synthesizing Realistic Workload Traces for Studying the Hadoop Ecosystem

Guanying Wang, Ali R. Butt, Henry Monti
Virginia Tech
{*wanggy,butta,hmonti*}@*cs.vt.edu*

Karan Gupta
IBM Almaden Research Center
guptaka@us.ibm.com

***Abstract*—Designing cloud computing setups is a challenging task. It involves understanding the impact of a plethora of parameters ranging from cluster configuration, partitioning, networking characteristics, and the targeted applications' behavior. The design space, and the scale of the clusters, make it cumbersome and error-prone to test different cluster configurations using real setups. Thus, the community is increasingly relying on simulations and models of cloud setups to infer system behavior and the impact of design choices. The accuracy of the results from such approaches depends on the accuracy and realistic nature of the workload traces employed. Unfortunately, few cloud workload traces are available (in the public domain). In this paper, we present the key steps towards analyzing the traces that have been made public, e.g., from Google, and inferring lessons that can be used to design realistic cloud workloads as well as enable thorough quantitative studies of Hadoop design. Moreover, we leverage the lessons learned from the traces to undertake two case studies: (i) Evaluating Hadoop job schedulers; and (ii) Quantifying the impact of shared storage on Hadoop system performance.**

***Keywords*-Cloud computing, Performance analysis, Design optimization, Software performance modeling**

I. Introduction

Cloud computing, powered by frameworks such as MapReduce [1], is emerging as a viable model for enabling fast time-to-solution for modern enterprise applications. The model has the potential to affect the IT industry in profound ways. Creating a MapReduce setup involves many performance critical design decisions, such as node compute power and storage capacity, choice of file system, layout and partitioning of data, and selection of network topology, to name a few. Moreover, a typical setup may involve tuning of hundreds of parameters to extract optimal performance. Estimating how applications would perform on specific cloud setups is critical, especially for optimizing existing setups and building new ones. To this end, simulation based approaches [2], [3] are becoming the main means for quickly and efficiently exploring the impact of design choices in cloud setups. These tools focus on studying how decisions about cluster design, run-time parameters, multi-tenancy and application design affect performance. A critical hurdle in exploring this design space is the lack of comprehensive, realistic workload characterizations and traces for driving the simulations. The results obtained from simulation studies are less valuable without realistic input traces. Unfortunately, such traces have been difficult to obtain and the performance and characteristics of commercial setups, such as Google's MapReduce [1], Hadoop [2], [4], Dynamo [5], and Quincy [6], remain shrouded in mystery. The lack of public traces prevents open-source research communities and academics from measuring the impact of their contributions on cloud systems, and thus hinders innovation.

Recently, cloud service providers, e.g., Google, have published some job traces [7], which raised hopes that more traces would soon become available. However, pertinent information such as the kind of applications the traces used, or even whether the traces are from a cloud system is unknown. Nonetheless, a careful examination of such traces, presented in this paper, shows that a number of lessons can be learned about the workload.

The main contribution of this paper is to describe the steps in analyzing available cloud traces to extract key workload characteristics. Moreover, the aim is to use this information to synthesize realistic cloud workloads, as well as allow for customizing the workloads for studying the role of different application configurations and settings.

Moreover, we leverage synthesized traces with realistic characteristics to perform two case studies of the impact of design changes on performance for one cloud system – Hadoop [8]. The first case study compares different job schedulers, and the second evaluates Hadoop design with Network Attached Storage (NAS) in contrast to the node-local storage used in standard systems.

In contrast to existing works on analyzing such traces [9], [10], which classify jobs based on resource usage, we take a binning approach where we classify tasks and jobs using time periods. This enables us to go beyond analysis and allows us to synthesize realistic cloud traces. We also employ an innovative approach where we compose tasks in a job into *slots* that let us infer how a job utilizes resources.

II. Approach

We are concerned with analyzing the distributed systems trace made public by Google, and extracting information that can be used for simulation of cloud setups. The trace is not known to be from a specific model such as MapReduce.

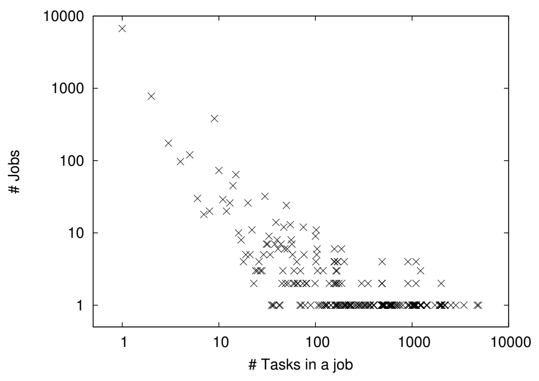

Figure 1. Distribution of number of tasks in a job.

However, our intuition is that the trace represents the kind of large-scale application workloads that can be supported through the cloud. Thus, this analysis can provide critical information for characterizing cloud workloads.

We considered several choices in analyzing the trace. Initially, we attempted to simply play back the traces in a simulator, e.g., MRPerf [3]. However, the duration, characteristics, and lack of detailed information in the traces prevented us from going this route. Instead, we dissect the trace to extract information such as job-length distribution, number of jobs, job and data dependencies, and computation-I/O ratios. This provides us with means to classify the jobs into different categories, as well as yields information about the different mix of jobs in a real trace. Next, we use the classification to infer cloud workload characteristics. Finally, we can use this information to synthesize new customized workloads that exhibit the same statistical characteristics as the original trace. Although the current synthesis process is manual, the lessons learned from the traces enabled us to study different design changes to Hadoop and their impact.

III. Overview of Available Trace

We use the trace from Google for our analysis. The trace spans a period of 370 minutes, with normalized processor and memory usage metrics collected every 5 minutes for a total of 74 intervals (timestamps). It represents $9,174$ jobs, with several sub-tasks each, for a total of $176,174$ tasks.

A. Job Statistics

Number of tasks per job: First, we examine the number of tasks associated with each job in the trace, as shown in Figure 1. It is observed that most of the jobs (84.6%) have a small number of tasks (< 5). Also, $6,713$ of the jobs have only one task. In contrast, a few jobs have a large number of tasks. E.g. the job with the most tasks has $4,880$.

Job Composition: We found that most jobs in the trace are such that all their associated tasks begin in an identical timestamp and end in another identical timestamp. The two timestamps may or may not be equal. All the tasks of such a job have the same length. We refer to these jobs as *synced jobs*. Conversely, jobs whose individual tasks start or finish during different timestamps are referred to as *non-synced* jobs. Figure 2 illustrates the different kinds of job compositions observed in the trace. Job (a) is a synced job, as all its tasks begin in the same timestamp 0, and end in the same timestamp 5. Job (b) is a job consisting of only one task, and is automatically a synced job. Job (c) is a non-synced job. Table I summarizes the numbers of each kind of tasks present in the trace. Note that 95% of all jobs in the trace are synced jobs. Also, about 5% of the jobs are non-synced jobs, which we discuss further in Section V.

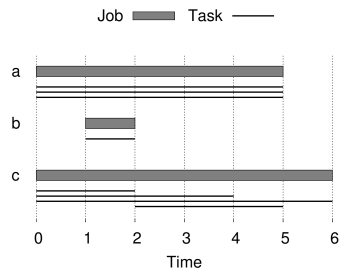

Figure 2. Kinds of jobs in the trace. (a) A *synced* job, (b) a one-task job, and (c) a *non-synced* job.

Table I
Different jobs making up the trace.

Job composition	Number	Percentage
Synced jobs	8738	95.2%
Jobs with only one task	6713	73.2%
Jobs with multiple tasks	2025	22.0%
Non-synced jobs	436	4.75%
Jobs that fits in slots	385	4.19%
Exceptions	51	0.56%
Total number of jobs	9174	100%

B. Task Statistics

Task execution length: We examine the execution duration of all tasks in the trace; Figure 3 shows the observed distribution. It can be observed that a large number of tasks $(6,341)$ run for only one timestamp, and $5,118$ of these tasks are associated with a single-task job. Moreover, since the trace represents a snippet of observed tasks, $60,823$ tasks are cut off either because they started earlier than the trace started, or because they did not finish before the trace ended. $35,191$ tasks are cut off at both ends, i.e., have an execution length of the entire 74 timestamps. We refer to these tasks as *full-length tasks*. Similarly, we define jobs that span the entire trace as *full-length jobs*.

Resource consumption of full-length tasks: Next, we examine the memory and CPU resources consumption of all the tasks, as shown in Figure 4. It is seen that the full-length tasks are responsible for a significant portion of both the CPU (76.5% on average) and memory (82.8% on average) consumed during the trace. Thus, even though the full-length jobs have unknown start and finish times, we retain them in our trace to ensure proper resource usage accounting.

Our examination of the trace shows that most jobs are

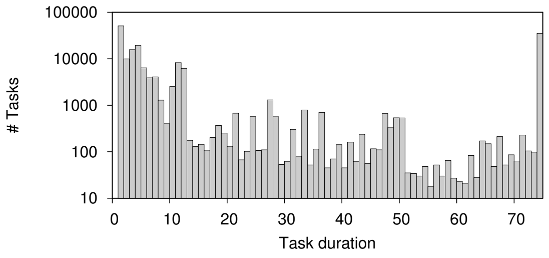

Figure 3. Distribution of duration of tasks in the trace.

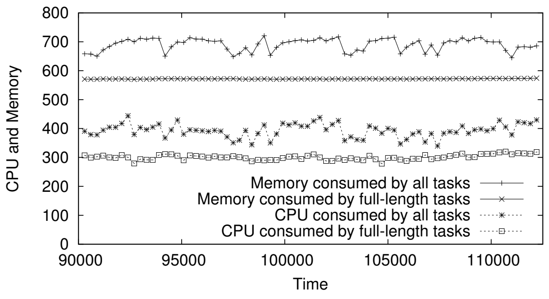

Figure 4. Normalized CPU and memory consumption of full-length tasks compared to all the tasks in the trace.

synced jobs and the full-length tasks are crucial as they consume a significant amount of CPU and memory resources. Moreover, although many jobs are small, a few large jobs have over a thousand tasks, run for a long time, and consume a large amount of resources.

IV. Trace Analysis

In this section, we examine the trace, especially the length of the tasks and jobs, in more detail and use this information to classify the tasks and jobs.

A. Classifying Tasks

We classify the tasks based on their execution duration. Specifically, we define three different categories. (i) *Seconds tasks* that run on the order of seconds up to one timestamp, i.e., 5 minutes. This category captures small tasks and bursty job behavior. (ii) *Minutes tasks* that run for more than 5 minutes but for less than 1 hour. (iii) *Hours tasks* that are long running tasks with a duration of more than 1 hour. This category captures longer jobs as well as *full-length* jobs. Table II(a) shows the number of tasks in each category.

Task categories over time: Next, we divided the trace into six hourly intervals. For each interval, we determined the number of tasks that fall into each of the three above categories. Figure 5 shows the results. It can be observed that the relative distribution of tasks in different categories does not change drastically from hour to hour. This is promising, as it allows for using the information from past intervals to predict how the trace would behave in the future intervals.

We analyzed the relative hourly task distribution further by examining the average and standard deviation of the number of tasks in each category over the six intervals. Next, we assumed a normal distribution for tasks and determined

Table II
(A) Classification of tasks based on duration. (B) Expected tasks category distribution for any hour.

(a)

Task category	Number
Seconds tasks	48085
Minutes tasks	74124
Hours tasks	15934
Full-length tasks	35202

(b)

Expected number
8014.2 ± 1084.2
15561.8 ± 2250.5
8463.2 ± 2156.8

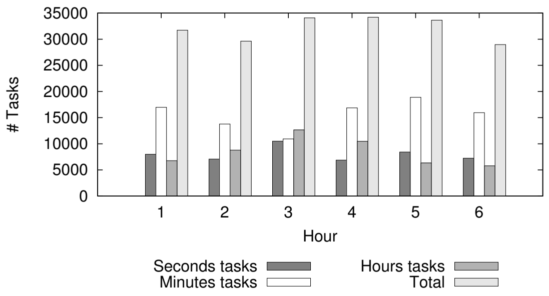

Figure 5. Distribution of different task categories over the duration of the trace.

the expected hourly distribution of the tasks in each category. Table II(b) shows the results with 95% confidence intervals. Although the *full-length* tasks are not predicted due to lack of duration data, a fixed number of *full-length* tasks can be added to a predicted distribution to synthesize a trace.

B. Classifying Jobs

In this section, we leverage the classification of tasks into different categories to classify jobs into different types. The goal is the same as it was for the classification of tasks, i.e., to determine the expected jobs behavior in future intervals based on past information.

For this purpose, we first divide the jobs into different types depending on the categories of tasks comprising the jobs. Table III(a) shows the number of jobs that contain different categories of tasks. The first part of the table shows jobs that contain only one category of tasks, followed by jobs that contain two different categories, and finally, jobs with all three categories of jobs are shown in the last part. We can see from the table that the majority of the jobs (96.4%) contain a single category of tasks only.

Next, similarly as for tasks, we examine the distribution of different types of jobs over hourly intervals. The result for this analysis is shown in Figure 6. The jobs with a single category of tasks are shown on the left, and the jobs with a mix of task categories on the right. Once again, we note that the relative hourly distribution of different types of jobs does not change drastically over the duration of the trace. Based on these statistics, we determine expected job type distribution for any hour based on 95% confidence intervals. The results are reported in Table III(b).

The classification of jobs and tasks provides crucial information that can be leveraged for synthesizing traces, which we discuss in subsequent sections.

Table III
(A) CLASSIFICATION OF JOBS BASED ON TASK CATEGORIES.
(B) EXPECTED JOBS TYPE DISTRIBUTION FOR ANY HOUR.

(a)

Job type	Number
Seconds tasks only	6370
Minutes tasks only	659
Hours tasks only	1815
Seconds+Minutes	167
Seconds+Hours	5
Minutes+Hours	90
Seconds+Minutes+Hours	68

(b)

Expected Number
1034.2 ± 147.0
134.0 ± 14.4
1759.0 ± 7.4
47.3 ± 4.2
5.0 ± 0.0
82.0 ± 3.1
58.2 ± 1.7

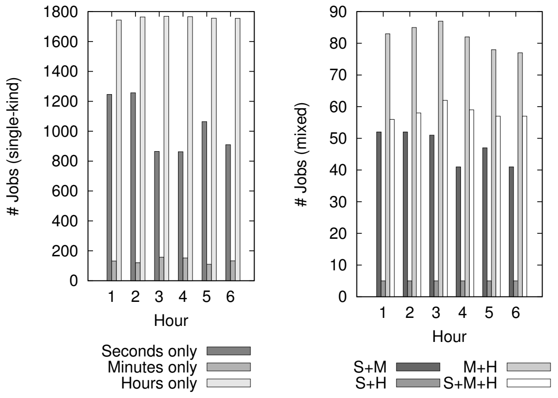

Figure 6. Distribution of different jobs types over the duration of the trace.

V. ANALYSIS OF NON-SYNCED JOBS

In this section, we focus on the 436 non-synced jobs in the trace (Table I). These jobs consist of tasks that run for more than a single timestamp and the individual tasks do not all share the same beginning timestamps and the same ending timestamps.

Slots: The actual unit of resource consumption in the trace is not known. Thus, for our analysis, we define the notion of a virtual resource as a "slot." Each task in one timestamp interval utilizes one slot, e.g., one job with three tasks running simultaneously would use three slots. If a task finishes, its slot may be assigned to another task. A job can have as many slots as it needs, which are assigned when needed. Slots serve as virtual resources, and knowledge about their utilization is helpful for understanding the execution pattern of a job. For example, the number of slots can be used to estimate the number of resources (e.g. in terms of processor cores) that a job requires. Moreover, by assigning the non-synced jobs to slots, we can visualize how the jobs are utilizing the resources. Such visualization can also provide more detailed information on the execution pattern of tasks in a job, e.g., we can infer whether tasks are reassigned or utilize the same slots until completion.

We examined all the non-synced jobs for their slot utilization over the duration of the trace. Figure 7 illustrates some of the interesting cases that were observed. Each graph represents a single job with multiple tasks. The Y-axis represents time from the start until the end of the trace. The X-axis represents the different slots assigned to a task. Thus, each bar in a slot represents a task in the job. A change in a color/pattern of a vertical bar shows that a task is finished and its associated slot is reassigned to a new task. White space in the graphs shows idle time.

Lets consider Figure 7(a) in more detail. The job starts with 46 tasks, each assigned to a different slot. Most tasks run for the entire length of the trace, hence the predominantly light green color observed in the graph. One task completes at about $11,100$ seconds, and the slot is reassigned to a different task as shown by the pink bar.

For different jobs, either their tasks fit almost perfectly into the slots (Graphs (a)-(h)) or not very well (Graphs (i)-(o)). However, some small white space may be due to the 5-minute granularity of the trace collection process.

A large number of jobs exhibit the patterns observed in Graphs (a)-(c), i.e., most of their associated tasks are full-length tasks. There are a few exceptions as observed by the different color/pattern bars in these graphs.

In contrast, Graphs (d)-(f) represent jobs where most slots are used by multiple tasks, and tasks in these jobs perfectly fill concurrent slots. Graphs (e) and (f) show predictable patterns. Further analysis shows that tasks in (e) all run for the same amount of time, and never overlap with each other. Graph (f) is only different from (e) in that (f) has 3 slots versus 1 of (e). Of course, both job (e) and (f) are cut off at both the beginning and the end of the trace, otherwise the patterns may have lasted for a longer time period. Some jobs, e.g., Graph (g), do not span the entire duration of the trace, but are also well arranged in slots. Graph (h) is a combination of many long running tasks and one slot that is used by many small tasks. Graph (i) shows an example of an exception to these observations. Tasks in Graph (i) all run for just 1 timestamp, and they do not fill the entire slot.

Graphs (j)-(l) are three different jobs that share the same irregular execution pattern. The graphs exhibit similar trends. A major difference between Graphs (j), (l), and (k) is that (j) and (l) have one full-length task, where as (k) has none. Despite this, the analysis and visualization points to common trends between these jobs. The intuition is that these jobs are very likely from the same application. Finally, Graphs (m), (n), and (o) represent jobs that have very irregular patterns.

From this visualization, we infer that most of the jobs for trace synthesis can be predicted using the information from the trace. Moreover, a small number of jobs can be added randomly to account for the observed irregular pattern jobs.

VI. EXAMPLE TRACE SYNTHESIS

The trace statistics provide us with critical information for synthesizing realistic cloud traces. While we can start from scratch for this synthesis, we focus on extending the original trace by several hours based on the observed statistics. Our algorithm for this purpose is shown as Algorithm 1. We first generate the total number of jobs based on number of jobs aggregated from Table III(b). Then for each job,

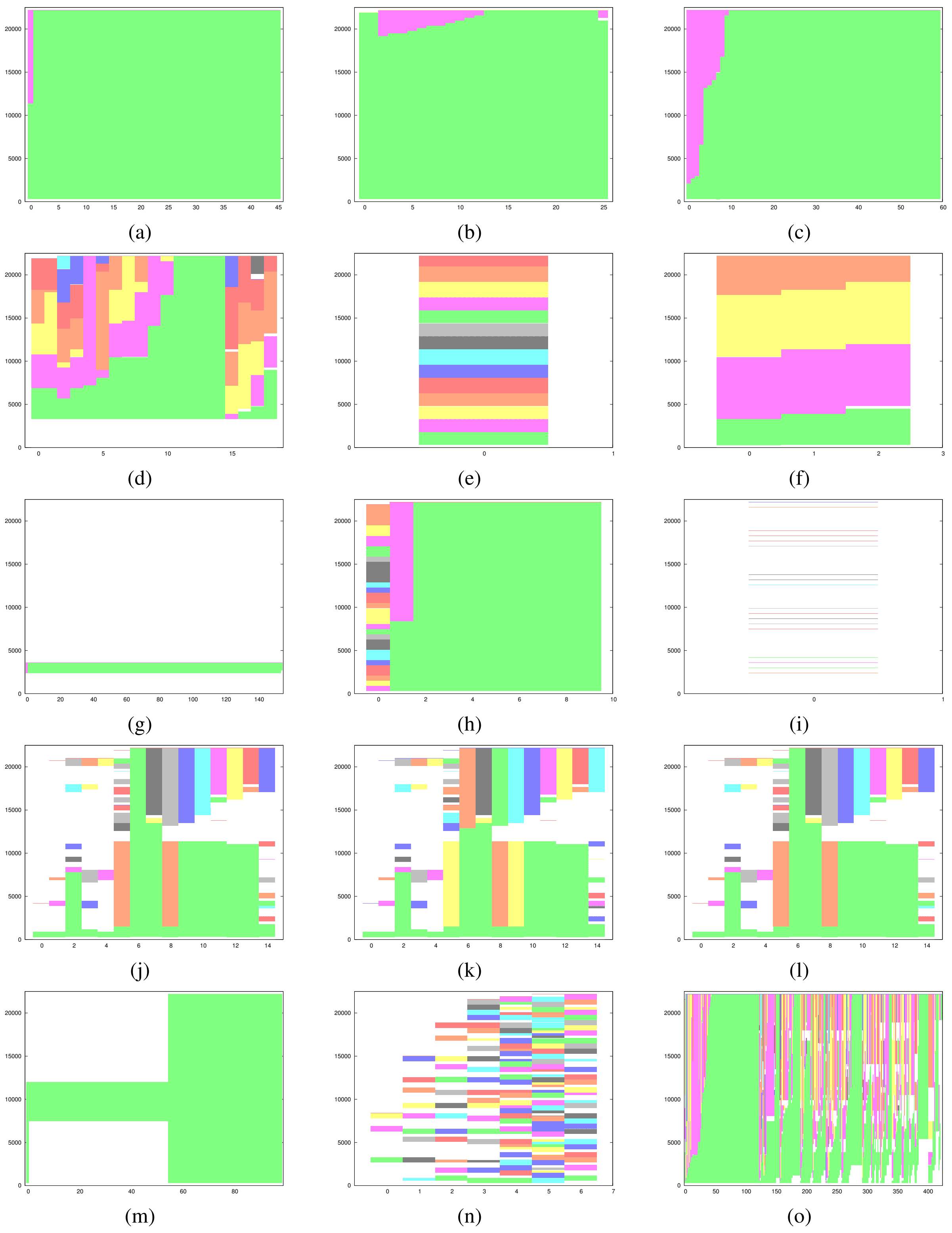

Figure 7. Example jobs showing slot utilization over duration of the trace. X-axis represents the slots assigned to the job. Y-axis represents the duration of the trace in seconds.

Algorithm 1 Steps for synthesizing realistic cloud traces.

```
Input: $duration$
  $total\_jobs = jobs\_per\_hour \times duration$ (Table III(b))
  for all $job$ in $total\_jobs$ do
    Determine whether $job$ is synced (Table I)
    if $job$ is synced then
      Generate $task\_count$ from (Figure 1)
      Generate $type$ and $length$ of the job (Table III(b))
    else
      Generate $task\_count$ from (Figure 1)
      Assign pattern (a)-(o) to $job$ (Figure 7)
    end if
  end for
```

we determine whether it is synced or non-synced based on distribution observed in Table I. The number of tasks in a job is then generated for a synced job, based on Figure 1. A synced job can be either a Seconds job, a Minutes job or an Hours job. So the type and length of a synced job can be generated from Table III(b). For a non-synced job, the number of tasks in a job is also generated based on Figure 1, with one-task jobs removed. Then a job is assigned a pattern similar to (a)-(o) in Figure 7, so tasks of the job can be generated. This algorithm yields a synthesized trace that exhibits similar statistical properties as the analyzed trace.

In the synthesis process, we have made several choices based on a random distribution. The reason for this is that a lot of crucial data is missing (or has been removed) from the trace available to us, such as information about I/O operations, consistency of the trace across multiple application runs, bursty nature of the workload that is not captured due to the granularity of trace collection intervals, and job and task dependencies. We believe that availability of this information will reduce the approximation of missing data points (that we had to employ) and enable us to create more accurate traces. Regardless of this, our analysis shows that the available trace serves as a good first step towards developing cloud workload generators.

VII. Applying Synthesized Traces: Case Studies

In this section, we present two case studies that are enabled by the availability of real or synthesized traces. We incorporated different Hadoop design changes in our MRPerf simulator [3], and then used the traces to drive MRPerf and analyzed the results. In the first case study, we evaluate different job schedulers for Hadoop tasks. In the second case study, we examine the impact of adding an extra NAS device to a Hadoop cluster on application performance. Both the case studies are based on our synthesized traces.

A. Background

1) MRPerf: MRPerf [3] is a discrete event simulator that simulates Hadoop [8] applications, and is a critical resource as Hadoop is used by larger enterprises including Yahoo! [11] and Facebook [4]. The original MRPerf simulator takes as input the topology of a cluster, the parameters of a job, and a data layout, and produces detailed simulation results about how the job would behave on the specified cluster configuration. In this work, we extended MRPerf to support our case studies, and used MRPerf as the platform to do experiments on. In the following, we detail the setup of our case-studies and how we collected and analyzed the results from the modified MRPerf.

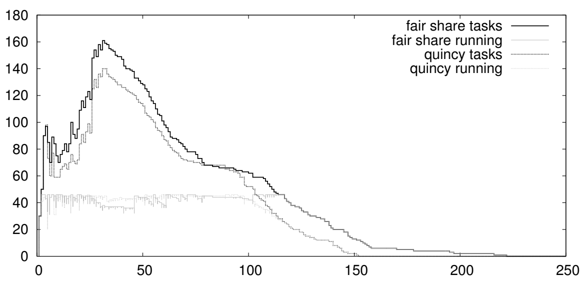

Figure 8. Job utilization under Fair Share and Quincy schedulers. The two bold lines on top show the number of map tasks that are submitted to the cluster, including running tasks and waiting tasks. Lower thin lines show the number of map tasks that are currently running in the cluster.

Table IV
Characteristics of different types of jobs.

Job type	cycles per byte	filter ratio
Terasort	40	1
Search	4–400	0–0.0001
Index	40	0.02–0.5
Compute	400–4000	1–10

B. Case Study I: Evaluating Hadoop Job Schedulers

1) Goal: Hadoop can run multiple jobs concurrently, and multiple scheduling algorithms [4], [6], [11] for Hadoop or similar systems have been proposed. To evaluate the effectiveness of different scheduling algorithms, we generate synthetic traces, and use the traces to drive MRPerf simulator. The traces contain four types of jobs, namely *Terasort*, *Search*, *Compute*, and *Index*. Table IV shows the description of these jobs. Columns "cycles per byte" and "filter ratio" are performance parameters used in MRPerf to characterize different applications. Please refer to the original MRPerf paper [3] for more details. The traces are then generated using a simple model with arrival times following a Poisson random process. We fix the maximum length of a time window T during which jobs may be submitted, and the rate $\frac{1}{\lambda}$ that jobs will arrive per second. On each arrival, a job of a random type is submitted in virtual time in the simulation. The type of a job is chosen among the four types with equal probability (25%), and parameters are randomly generated if necessary. The trace is generated for time period T. The expected number of jobs that will be generated in a trace is $\frac{T}{\lambda}$. MRPerf is then driven by this trace.

The virtual cluster we modeled in MRPerf simulator is a

Table V
LOCALITY OF ALL TASKS UNDER FAIR SHARE AND QUINCY.

Locality	Fair Share	Quincy
Data-local	167	304
Rack-local	131	0
Rack-remote	6	0

Table VI
LOCALITY OF ALL TASKS IN DIFFERENT TRACES.

Locality	*Terasort*		*Compute*	
	Fair Share	Quincy	Fair Share	Quincy
Data-local	440	652	258	361
Rack-local	198	0	96	2
Rack-remote	14	0	9	0

24-node cluster organized in two racks. The two racks are connected over a 8 Gbps interconnection, and the network bandwidth within a rack is 1 Gbps. We choose rate ($\frac{1}{\lambda}$) as 1 (job per second) so that all jobs are submitted towards the beginning of a trace, and the cluster quickly becomes fully utilized and will remain so until most jobs are finished.

2) MRPerf Modification: We implemented the naive Fair Share scheduler [4] in MRPerf simulator. The delay scheduling in the Fair Share scheduler is not implemented because the length of the traces is too short to reflect the advantage of delays. We also implemented the Quincy [6] scheduler, and ported it to Hadoop. We only studied the non-preemptive Quincy scheduler since the Fair Share scheduler does not support preemption. Since Quincy achieves overall optimal locality, but naive Fair Share without delay scheduling does not, Quincy is expected to perform better than Fair Share.

3) Evaluation: We generate a trace with 28 jobs, and run the trace under Fair Share and Quincy. Table V shows the locality of tasks under both schedulers. We denote the node that a task runs on as the worker, and the node with data as the host. Data-local means that the worker and host are the same node. Rack-local means the worker and host are not the same nodes, but they are in the same rack. Rack-remote means the worker and host are in different racks. Quincy achieves perfect locality, much better than Fair Share. Figure 8 shows the overall utilization of the cluster under Fair Share and Quincy. Bold lines on top show the number of map tasks that are submitted to the cluster, including running tasks and waiting tasks. Lower thin lines show the number of map tasks that are currently running in the cluster. Solid lines show total tasks and running tasks for Fair Share, dashed lines show total tasks and running tasks for Quincy. This figure confirms the advantage of Quincy over Fair Share. Although driven by the same trace, Quincy achieves better data locality and finishes tasks faster, so Quincy finishes earlier overall.

Furthermore, we also use the same framework to study the impact of data locality on different types of jobs. Instead of a trace with mixed types of jobs, we generated four traces, each of which consists of only one type of job. Figure 9 and Figure 10 show results for traces of *Terasort* and *Compute* jobs, respectively. Results from *Search* and *Index* are omitted

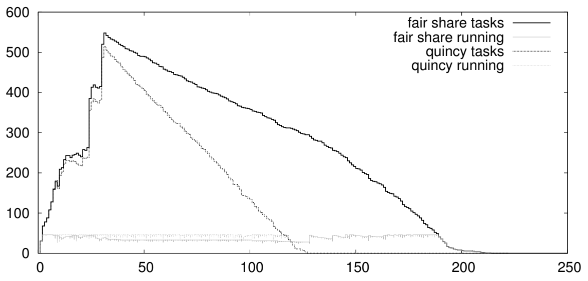

Figure 9. Job utilization of *Terasort* trace under Fair Share and Quincy.

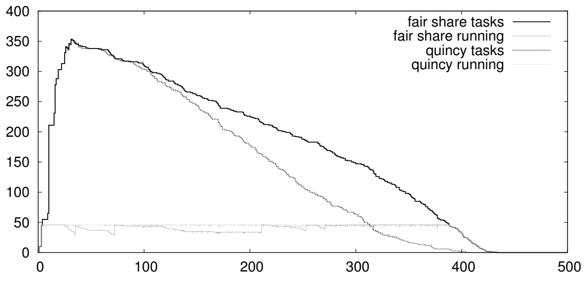

Figure 10. Job utilization of *Compute* trace under Fair Share and Quincy.

since they are similar to *Terasort*. Since *Compute* jobs involve heavy computation, the overall completion times are not significantly different under Fair Share and Quincy. A much larger difference can be observed for *Terasort* jobs. Table VI shows locality under both schedulers. Similar locality is achieved for both traces. Therefore, we conclude that *Compute* jobs are less affected by locality.

C. Case Study II: Evaluating the Role of NAS on Hadoop Cluster Performance

1) Goal: In this case study, we focus on the performance impact of a Network-Attached Storage (NAS) device added to a Hadoop cluster on application performance. We assume that the device is used for storing the input/output data of Hadoop jobs, which usually sit on the Hadoop Distributed File System (HDFS) and are distributed across all nodes. Replacing or augmenting HDFS with NAS could provide many benefits in practice, including ease of management, reliability, reduced cost, backward compatibility, etc. The trade-off between NAS versus HDFS is out of scope of this paper. Rather, we focus on the performance impact of adding an extra NAS device in a Hadoop cluster. Figure 11 shows a common Hadoop cluster, and Figure 12 shows a Hadoop cluster with NAS. In the following discussion, we will use the word "*Local*" to refer to a common Hadoop cluster, and the word "*NAS*" to refer to Hadoop cluster with NAS device. *Local* and *NAS* refer to where the data is stored.

Since reading from and writing to NAS are always rack-remote, and the NAS device is shared by all nodes, the I/O performance of *NAS* will not be optimal for data-intensive applications. Our aim is to investigate how the performance of *NAS* is related to the number of racks serviced by a NAS

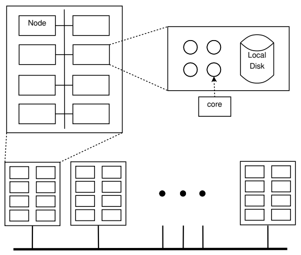

Figure 11. Architecture of a typial Hadoop cluster.

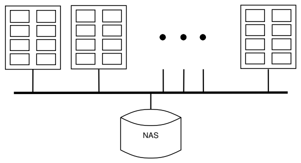

Figure 12. Architecture of a Hadoop cluster with a NAS device.

Table VII
SLOW DOWN FACTOR OF *NAS* VERSUS *Local* IN DIFFERENT CONFIGURATIONS.

Scenario	*Terasort*	*Search*	*Compute*	*Index*
2-rack 15-job	1.73	1.57	1.21	1.31
2-rack 30-job	1.69	1.52	1.05	1.68
4-rack 15-job	1.69	2.11	1.31	1.87
Faster NAS	1.34	1.11	1.04	1.02
Lost Locality (60%)	1.07	1.47	**0.92**	1.37
Increased Replicas	1.25	1.63	**0.83**	1.37

device, and to the overall workload. Moreover, for certain types of compute-intensive applications, *NAS* may be good enough to match or surpass the performance of *Local*. In our experiments, different types of applications and the impact of NAS on their performance will be examined and compared.

2) MRPerf Implementation to Enable Integration of NAS with Hadoop: We implemented a new kind of device in MRPerf, the NAS device, to enable this case study. The NAS device is modeled as a storage device with a certain capacity, and performance parameters including read and write I/O bandwidth. To simulate applications in a Hadoop cluster with a NAS device, input data must be placed on the NAS device instead of on local disks of nodes. Schedulers considering locality must be aware that reading data from the NAS device is always rack-remote. Moreover, output data must be written to NAS, not local disks on the node where a reduce task is run. Note that intermediate map and reduce task data is still stored on local disks.

3) Evaluation: The base configuration that we have employed for our experiments in this section is a 2-rack topology with 15 jobs (2-rack 15-job). The topology is similar to the topology used in Case Study I. An extra NAS device is added and connected to the core network via a 2 Gbps link. We also looked at other configurations including 2 racks with 30 jobs (2-rack 30-job) and 4 racks with 15 jobs (4-rack 15-job). In each configuration, we generate four traces, each of which contains only one type of job. Then we drive the simulator with the four traces, and observe the difference in results. The results show which types of applications are more friendly to *NAS*, and which types of applications suffer severe performance degradation with *NAS*. We employ Fair Share as the scheduler for both *Local* and *NAS* for a fair comparison.

The results are presented in Table VII. In the base case (2-rack 15-job), all workloads suffer when using *NAS* compared to *Local*. Among all workloads, *Compute* achieves the lowest slowdown factor, because the input and output data are much larger than the other cases. Next we tried to decrease the average load on the storage by increasing the number of racks to 4. In the 4-rack 15-job configuration , however, we see that all workloads suffer greater slowdown than the base case. This occurs because the nodes read from and write to local disks in *Local*, and the load on local disks is smaller. In contrast, load on the NAS device is not changed since number of jobs remains the same. Results of the 2-rack 30-job configuration shows the opposite approach, increasing average load on every node. *Compute* trace achieves lower slowdown, while *Index* slowdown is higher, and *Terasort* and *Search* remain the same. As we checked the simulation results in detail, we found that *Compute* in *Local* is stretched by a single long job. If the trace were without the job, *Compute* in *Local* would finish earlier, and *NAS* slowdown factor would be higher. Therefore, we conclude that *NAS* slowdown is related to the number of racks, not to the number of jobs. The more racks one NAS device supports, the higher slowdown factor will become.

4) Extreme Cases Where NAS *Would be Preferable:* To demonstrate that *NAS* may be beneficial in at least some cases, we come up with several extreme situations where *NAS* could get better results over *Local*. Although these situations are less likely to occur in real world, they do help in understanding the trade-offs to consider when employing *NAS* versus *Local* in Hadoop clusters.

Faster NAS Device: A straight-forward way to boost *NAS* performance is to increase the performance of the NAS device. The default NAS I/O bandwidth is 1000 MB/s for read and 300 MB/s for write, and it is connected via a 2 Gbps link. We increased the I/O bandwidth of the NAS device to 3000 MB/s for read and 1000 MB/s for write. We also increased the bandwidth of the link connecting it to the router to 8 Gbps. As a result, the NAS device is turned into a roughly 3x-4x faster device. The performance of such *NAS* configuration will be better than the original *NAS*, if not better than *Local*.

Comparing to the base case (2-rack 15-job) , we can see that each type of application has a lower slowdown factor. *NAS* performance of the *Compute* and *Index* applications almost match *Local*'s performance. *Search* also achieves a lower slowdown factor. *Terasort* seems to be the application that faster NAS does not help. Just by increasing the performance of the NAS device, the performance of some applications could catch up with the performance in *Local*, but our tentative results show that *NAS* may never surpass *Local* even if the NAS device is very fast.

Lost Locality: Another situation we examined is Lost Locality in *Local*. Since *Local* can achieve good locality and hence good performance, we are interested in how *Local* would perform if part of the locality is lost. We

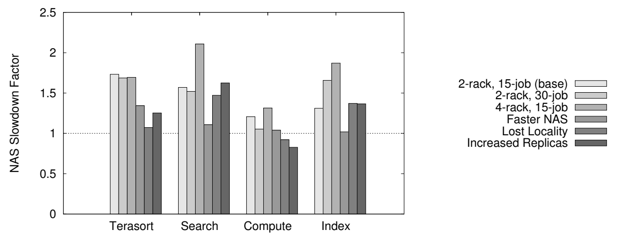

Figure 13. *NAS* slowdown factors in all cases studied.

implemented an artificial scheduler, which on probability p gives up locality by scheduling tasks arbitrarily (random scheduler), and with remaining probability $1-p$ schedules tasks with best achievable locality. By tweaking p between 0% and 100%, the artificial scheduler works like a mix of Fair Share and a random scheduler.

The *NAS* slowdown factor is calculated against *Local* with $p = 60\%$. The meaning of the experiment can be interpreted as following. If 60% locality is lost, *Local* will likely perform worse. In this case, how will *NAS* compare to *Local*? The *Compute* trace performs better in *NAS* than in *Local*. This is the first case where *NAS* performs faster than *Local*, although under unusual assumptions. *Compute* jobs are heavy in terms of computation as well as the size of output data. With a faster NAS device, *Compute* jobs can write over network to *NAS* faster than to remote disks.

Increased Replicas: In our default implementation, the output of every reduce task is put on local disk, and not replicated on other nodes. Support for multiple replicas was not implemented due to an issue in legacy code. The current MRPerf simulator has evolved so that we can support multiple replicas. We change *Local* so that every reduce task writes out 3 replicas of the same data on randomly selected nodes like Hadoop does. Writing multiple copies on different nodes incurs overhead. However, in *NAS*, since everything goes to NAS, writing multiple copies is not required. By comparing *NAS* against *Local* with multiple replicas, we hope to see *NAS* perform better than *Local* which suffers from the overhead of creating replicas.

Results show that *Compute* performs better in *NAS* than in *Local* with replicas. *Terasort* achieves 125%, a closer gap between *NAS* and *Local*, compared to the base case (173%). *Compute* and *Terasort* both output large amounts of data, and *Compute* more so than *Terasort*. Therefore, *Compute* achieves the largest improvement and *Terasort* a smaller improvement. We conclude tentatively that in Hadoop clusters where output data is replicated, applications with large amounts of output could benefit from *NAS*.

Figure 13 shows all slowdown factors presented in Table VII. Apparently *Compute* benefits the most from *NAS*. *Compute* jobs are heavy in terms of computation. Therefore, computation-intensive jobs will likely benefit from *NAS*. Among all cases, Faster NAS provides the best overall performance improvementm, though it does not surpass *Local*. It works across all workloads.

VIII. Conclusion

We have presented an initial analysis of a distributed computing trace released by Google. We found that jobs can be classified into different types based on their execution duration and the categories of associated tasks, and that the tasks exhibit specific usage patterns. We are also able to predict, within narrow 95% confidence intervals, hourly relative distribution of the types of jobs. Manual synthesis also revealed that these characteristics can be used to build realistic workload traces, which can then be used to drive cloud setup simulators. We leveraged the lessons learned from the traces to undertake two Hadoop design case studies and found that: (i) scheduling for better locality indeed yields better overall performance; and (ii) compute-intensive applications are less sensitive to locality and can benefit more from a design that uses an extra NAS device.

Acknowledgment

This material is based upon work supported by: the NSF under Grants CCF-0746832 and CNS-1016793; an IBM SUR Award; and a NetApp Faculty Fellowship.

References

[1] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” in *Proc. USENIX OSDI*, 2004.

[2] A. C. Murthy, “Mumak: Map-Reduce Simulator,” MAPREDUCE-728, Apache JIRA, 2009. [Online]. Available: http://issues.apache.org/jira/browse/MAPREDUCE-728

[3] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, “A Simulation Approach to Evaluating Design Decisions in MapReduce Setups.” in *Proc. IEEE MASCOTS*, 2009.

[4] 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 *Proc. EuroSys*, 2010.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[6] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy: fair scheduling for distributed computing clusters,” in *Proc. SOSP*, 2009.

[7] J. L. Hellerstein, “Google Cluster Data,” 2010. [Online]. Available: http://googleresearch.blogspot.com/2010/01/google-cluster-data.html

[8] Apache Software Foundation., “Hadoop,” May 2007. [Online]. Available: http://hadoop.apache.org/core/

[9] 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.*, vol. 37, pp. 34–41, March 2010.

[10] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz, “Analysis and lessons from a publicly available google cluster trace,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-95, Jun 2010.

[11] A. C. Murthy, “The hadoop map-reduce capacity scheduler,” 2011. [Online]. Available: http://developer.yahoo.com/blogs/hadoop/posts/2011/02/capacity-scheduler/