Using Realistic Simulation for Performance Analysis of MapReduce Setups

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

Prashant Pandey, Karan Gupta
IBM Almaden Research Center
{ppandey, guptaka}@us.ibm.com

ABSTRACT

Recently, there has been a huge growth in the amount of data processed by enterprises and the scientific computing community. Two promising trends ensure that applications will be able to deal with ever increasing data volumes: First, the emergence of cloud computing, which provides transparent access to a large number of compute, storage and networking resources; and second, the development of the MapReduce programming model, which provides a high-level abstraction for data-intensive computing. However, the design space of these systems has not been explored in detail. Specifically, the impact of various design choices and run-time parameters of a MapReduce system on application performance remains an open question.

To this end, we embarked on systematically understanding the performance of MapReduce systems, but soon realized that understanding effects of parameter tweaking in a large-scale setup with many variables was impractical. Consequently, in this paper, we present the design of an accurate MapReduce simulator, *MRPerf*, for facilitating exploration of MapReduce design space. *MRPerf* captures various aspects of a MapReduce setup, and uses this information to predict expected application performance. In essence, *MRPerf* can serve as a design tool for MapReduce infrastructure, and as a planning tool for making MapReduce deployment far easier via reduction in the number of parameters that currently have to be hand-tuned using rules of thumb.

Our validation of *MRPerf* using data from medium-scale production clusters shows that it is able to predict application performance accurately, and thus can be a useful tool in enabling cloud computing. Moreover, an initial application of *MRPerf* to our test clusters running Hadoop, revealed a performance bottleneck, fixing which resulted in up to 28.05% performance improvement.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: *modeling techniques*; I.6.3 [**Simulation and Modeling**]: Applications

General Terms

Experimentation, Verification

Keywords

Cloud Computing, Hadoop, Simulation, MapReduce

1. INTRODUCTION

Recent trends show that cloud computing is becoming a robust paradigm for supporting modern data-intensive enterprise applications. Cloud computing aims to provide efficient resource utilization and improved performance, as well as ease-of-use by freeing the application developers from issues of resources scheduling, allocation, and associated data management. In this context, the MapReduce [14] programming model has emerged as an important means of instantiating cloud computing and simplifying application development. However, the configuration design-space of MapReduce has not been studied in detail. This is a complex problem as a typical MapReduce configuration can encompass hundreds of parameters, e.g., node configuration (number of disks and compute capacity), network topology (inter and intra-rack), choice of file system, data partitioning and layout, types of schedulers, etc. – all of which affect application performance. Moreover, empirical insights for certain specific configurations, e.g., Google's MapReduce setup [13], cannot be simply extended to other setups, and no tool or model is available to the community for studying MapReduce application performance. This leaves the users guessing as to how their applications will behave under particular configurations, which is especially crucial for building new systems, where correct performance estimates are required.

In this paper, we explore how choices about cluster design and run-time parameters affect application performance. The scale of the system, as well as the large turn-around time for realizing a given configuration, precludes using actual machines for design space exploration. Thus, we develop an accurate MapReduce simulator, *MRPerf*, to facilitate performance analysis. The goal is to comprehensively capture the various design parameters of MapReduce. MapReduce applications run on large clusters, so their performance depends on the interaction of many factors. The insights gained through *MRPerf* will be useful in comprehending these factors. We expect *MRPerf* to be used by researchers and practitioners to understand how their MapReduce applications will behave on a particular configuration, and how they can improve the applications and platforms to optimize performance.

1.1 Motivation & Challenges

MapReduce [14] is a simple model for machine-independent parallel programming at large scales. It provides minimal abstractions, hides architectural details, and supports transparent fault tolerance. MapReduce is ideal for massive data searching and processing operations. It has shown excellent I/O characteristics for traditional clusters. Current trends show that MapReduce is considered a high-productivity alternative to traditional parallel programming paradigms for enterprise computing [13, 9] as well as peta-scale scientific computing [18, 7].

Although the use of MapReduce is becoming wide-spread, it is not well understood how its publicly-available implementations, e.g., Hadoop [9] and others [7, 3], perform for specific configurations and applications. In fact, a quick survey of related discussion forums [4] reveals that most users are relaying on rules-of-thumb and in-exact science; for example it is typical for system designers to simply copy/scale another installation's configuration without taking into account their specific applications' needs. *MRPerf* aims to answer questions being asked by the community about MapReduce setups: How well MapReduce scale as the cluster size grows to extreme scales, e.g., 10,000-nodes? Can a particular cluster setup yield a desired I/O throughput? In addition, *MRPerf* can be used to understand the sensitivity of application performance to platform parameters, network topology, node resources and failure rates. The scale and complexity of a MapReduce setup results in a deluge of parameters that should be tuned, tested, and evaluated to yield an efficient system.

A key challenge in designing *MRPerf* is determining the right level of component abstraction: If every component is simulated thoroughly, it may take prohibitively long to produce results; conversely, if important components are abstracted out, the results may not be accurate. Moreover, the performance of a MapReduce application depends on the data layout within and across racks and the associated job scheduling decisions. Therefore, it is essential to make *MRPerf* layout-aware and capable of modeling different scheduling policies. Furthermore, the shuffle/sort and reduce phases of a MapReduce application are dependent on the input and require special consideration for correct simulations. Finally, a simulator is valuable only if its results can be verified on (some) real setups. This is challenging as verifying *MRPerf* at scale requires access to a large number of resources, and setting the resources up under different network topologies, per-node resources, and application behaviors.

The goal of our simulator is to take on these challenges and answer the above questions. Moreover, we aim to address issues such as determining optimal configuration for a MapReduce infrastructure given a fixed budget, selecting how much disk space should be allocated per node, and determining the optimal ratio of computing power to disk space per rack. Studying such questions on real hardware is next to impossible given limited budgets, and the complexity of the system precludes correct analytical models. Therefore, we opt for building a simulator for this study.

1.2 Contribution

In this paper, we present the design and evaluation of a realistic simulator for the widely-used MapReduce implementation, Hadoop [9]. Specifically, this paper makes the following contributions:

- Design, develop, and implement an accurate Hadoop simulator, *MRPerf*, that provides performance estimates for many classes of applications;
- Verify, fine-tune, and improve the simulator by performing measurements on actual (small to medium-scale) Hadoop setups; and
- Apply *MRPerf* to identify and address performance and configuration bottlenecks in Hadoop.

Our validation using a 40-node (320-core) cluster shows that *MRPerf* can accurately model MapReduce setups running Hadoop. Moreover, our evaluation revealed a performance bug in Hadoop, fixing which resulted in up to 28.05% improvement in performance for a typical sort application.

2. MAPREDUCE APPLICATION PERFORMANCE

The performance of a MapReduce application on a cluster depends on a large number of factors. In the following, we categorize and discuss these factors.

The MapReduce framework automatically parallelizes applications[1] and utilizes available compute, storage and communication resources to execute them. The exact manner in which a job gets split, and when and on what resources tasks are executed is influenced by a variety of *configuration parameters*, and is an important determinant of performance. Examples of these parameters include:

- *Data replication factor*: More replication makes scheduling decisions easier, but associated data writes and initial data ingestion into the cluster become slower.
- *Data block (or chunk) size used by the storage layer*: Chunk size affects the amount of data processed by a single map job. A trade-off must be made between amortizing task startup and disk seek times (with large blocks) and creating the maximum opportunity for parallelism (using small blocks).
- *Number of map and reduce tasks in a job*: These numbers affects CPU, network and disk utilization. The trade-off lies in the observation that efficiently utilizing different resource setups may require different settings.

In addition to configuration parameter, the infrastructure characteristics also impact performance. The infrastructure on which MapReduce executes typically involves a large number of machines arranged as follows. A group of compute nodes with several disk(s) connected together make up what is referred to as a *rack*. Each node in a rack is usually a single network hop away from every other node. Multiple racks are connected to each other using a hierarchy of switches to create the cluster. We refer to the node capabilities and network topology as *cluster parameters*, and different design choices in this context have a huge impact on performance. Examples of the such choices include:

- Characteristics of CPU, RAM, and disk on each node.
- Heterogeneity of nodes making by a rack, as well as heterogeneity of racks making up a cluster.

[1]Applications that exhibit embarrassingly parallel behavior.

- Connectivity between nodes within a rack.
- Inter-rack connectivity and topology across racks.

Moreover, design and implementation choices within a MapReduce framework also affect application performance. Decisions about data placement and task scheduling are particularly important. We group these choices together as *framework parameters*. Examples of these parameters include:

- Data placement algorithm, which decides where to place data blocks – a good placement algorithm makes it easier to schedule tasks near to where their associated data is stored.
- Task scheduling algorithm, which decides where to place tasks (in relation to their data) and whether to schedule redundant jobs preemptively.
- Data movement policy, which dictates how data is moved between job phases.

These factors interact in complex ways to affect the performance of applications. The behavior of the application in terms of its disk, CPU and network usage in different stages of execution causes the impact of a particular factor to vary. For example, the connectivity between nodes is not an important factor for a job that produces little output if the map tasks are scheduled on nodes that hold the input data. But, for the same application, if the scheduler is not able to place jobs near the data (e.g. if the data placement is skewed), then network bandwidth between the data and compute nodes might become the limiting factor in application performance. The complexity and vast range of interactions between application behavior and configuration, cluster and framework parameters has convinced us that an analytical model would be extremely complicated (or very inaccurate), and a simulator would be a better choice that is able to capture the performance of applications on such systems more accurately.

3. SIMULATOR DESIGN

To address the challenges faced in designing large MapReduce setups, we have implemented a prototype simulator for Hadoop [9], *MRPerf*. In the following section, we describe the simulator design in detail.

3.1 Architecture Overview

MRPerf provides fine-grained simulation at sub-phase level, models inter- and intra-rack network communications, as well as activities inside a single node, such as processor time consumed by a job, and disk I/O time for reading inputs and writing results. The simulator takes several files as input, including node specification, cluster topology, data layout, and job description. The output is a detailed trace, which provides the job execution time, the amount of data transferred, and the time-line of each phase of the task. The output trace can also be visualized for analysis. The current implementation is limited to modeling a single storage device per node, supporting only one replica for each chunk of data in HDFS, and not modeling certain optimizations such as speculative execution. However, lack of such support does not restrict *MRPerf*'s ability to model performance of most Hadoop setups. Moreover, removing these limitations is the focus of our ongoing research.

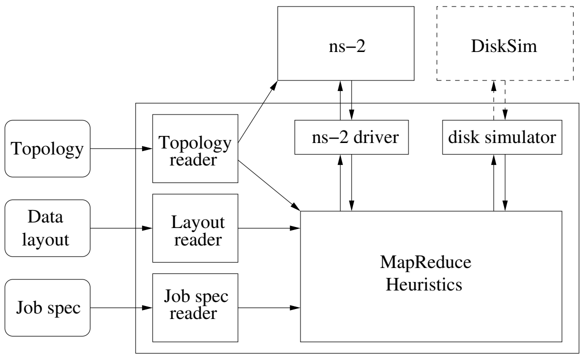

Figure 1: *MRPerf* architecture.

Figure 1 shows the overall architecture of *MRPerf*. The input configuration is provided in a set of files, and processed by different processing modules (*readers*), which are also responsible for initializing the simulator. The ns-2 driver module utilizes the ns-2 [2] network simulator to realistically simulate network traffic. Similarly, the Disk module provides modeling for the disk I/O. We note that, although we use a simplistic disk model, this module can be easily extended to include advanced disk simulators such as DiskSim [1]. All of this information is then used to drive the MapReduce Heuristics module that simulates Hadoop's behavior.

The simulator starts by reading all the configuration parameters, and instantiating the required number of nodes arranged according to the specified topology. Each node starts to *run* the job scheduled for it, and sends and receives associated messages through ns-2.

A node's resources, i.e., processors and disks, are shared among tasks assigned concurrently to the node. In real settings, I/O and computation are overlapped given asynchronous prefetching present on most modern Hadoop nodes. In contrast, *MRPerf* does not overlap I/O and computation assigned to a thread, rather divide these tasks into distinct sequential phases. However, I/O and computation across threads and processors are overlapped. This results in a simplified simulator design, at the cost of some accuracy. However, as our results show, this approach provides fairly accurate results.

3.2 Input Specification

MRPerf requires input from the user, which can be classified into three parts: cluster topology specification, application job characteristics, and the layout of the application input and output data. *MRPerf* relies on ns-2 for network simulation, thus, any topology supported by ns-2 is automatically supported by *MRPerf*. The topology is specified in XML format, and is translated by *MRPerf* into tcl format for use by ns-2. Example 1 shows a sample topology specification.

To capture job characteristics, we assume that a job has simple map and reduce tasks, and that the computing requirements are dependent on the size, and not content, of the data. For accuracy, several sub-phases within a map task are modeled separately, e.g., JVM start, single or multiple rounds of map, sort and spill, and a possible merge. Compute time for each data-size-dependent sub-phase is captured using a cycles/byte parameter. Thus, a set of cycles/byte measured for each of the sub-phase provides a mean for specifying application behavior. Some application phases do not involve input-dependent computation, rather fixed overheads, e.g., connection setup times. These steps are captured by measuring the overhead and using it in the simulator. This approach to capture overall application behavior is in-line with what we have observed in real experiments, and as we show later, is effective. Example 2 shows a sample job specification.

Finally, the data layout provides the location of the main node handling the job metadata, the location of actual data on the simulated nodes, replication factor, etc. Data layout affects data-computation co-location in the map phase, and thus the overall performance. Finally, we assume that job output data is proportional to the input data and thus no separate means for modeling the job output data is required. Example 3 shows a sample simulation specification.

```
<topo>
  <machine_type> ... </machine_type>
  <machine_type> ... </machine_type>
  <switch_type> ... </switch_type>
  <rack_group>
    <compute_node_group>
      <machine>Demo Cluster Spec</machine>
      <node_index>00</node_index>
      <node_index>01</node_index>
      <node_index>02</node_index>
      <node_index>03</node_index>
    </compute_node_group>
    <switch>
      <switch>Demo switch</switch>
      <switch_index>1</switch_index>
    </switch>
    <rack_index>1</rack_index>
    <rack_index>2</rack_index>
    <name>rg1</name>
  </rack_group>
  <router>
    <connect_to_group>
      <rack_group_name>rg_rg0</rack_group_name>
      <switch_index>1</switch_index>
    </connect_to_group>
    <name>r1</name>
  </router>
</topo>
```

Example 1: Topology specification.

Some of these parameters are function of the physical cluster topology, while others can be collected by profiling a small-scale MapReduce cluster or running test jobs on the target cluster.

3.3 Integration with ns-2

Given the established use of ns-2 [2] in realistically simulating networked systems, *MRPerf* is built on top of ns-2. We employ packet-level simulation to get accurate network behavior. In addition to using ns-2 services, *MRPerf* simulates the map and reduce tasks, manages their associated input and output, make scheduling decisions, and models disk and processor load. The reliance on ns-2 ensures that our simulator is able to capture the effects of the interaction between various components at fine granularity.

Each simulated node has several processors and a single disk, and the processing power is divided equally between the jobs scheduled for the node. One restriction is that a task

```
<job>
  <jvm_start_cost>5.0*1000*1000*1000</jvm_start_cost>
  <map>
    <cycles_per_byte>20</cycles_per_byte>
    <sort_cycles_per_byte>50</sort_cycles_per_byte>
    <merge_cycles>1.0*1000*1000*1000</merge_cycles>
    <filter_ratio>
      <uniform>
        <min>0.5</min>
        <max>1</max>
      </uniform>
    </filter_ratio>
  </map>
  <reduce>
    <merge_cycles>5.0*1000*1000*1000</merge_cycles>
    <cycles_per_byte>20</cycles_per_byte>
    <filter_ratio>
      <uniform>
        <min>1</min>
        <max>1</max>
      </uniform>
    </filter_ratio>
  </reduce>
  <average_record_size>10</average_record_size>
  <job_tracker>n_rg0_0_ng0_1</job_tracker>
  <name_node>n_rg0_0_ng0_0</name_node>
  <input_dir>data</input_dir>
  <output_dir>output</output_dir>
</job>
```

Example 2: Job specification.

```
<layout>
  <dir name="data">
    <file name="file_00000000">
      <chunk id="0">
        <rep>d_rg0_0_ng0_0_disk0</rep>
        <rep>d_rg0_0_ng0_1_disk0</rep>
      </chunk>
      <chunk id="1">
        <rep>d_rg0_0_ng0_2_disk0</rep>
      </chunk>
    </file>
    <file name="file_00000001">
      <chunk id="0">
        <rep>d_rg0_0_ng0_0_disk0</rep>
        <rep>d_rg0_0_ng0_2_disk0</rep>
      </chunk>
    </file>
  </dir>
</layout>
```

Example 3: Data layout.

cannot be split among processors, thus if there are fewer jobs scheduled for a node than the number of available processors, some of the processors on that node will remain idle. This is in-line with how tasks are executed in real setups. Also, each simulated node is responsible for tracking its own processor and disk usage, and other statistics.

The *MRPerf* kernel works as follows. First, *MRPerf* creates a number of simulated nodes using the standard ns-2 interface, and configures ns-2 to connect the nodes in the specified topology. Each simulated node is instantiated, in essence, using a set of call-back functions that are triggered when different messages are received. This approach leverages the `TcpApp` code in ns-2. Next, an `init` message is sent to all the nodes, which results in the nodes simulating disk and processor use, as well as sending out interaction messages, consequently triggering further action and advancing

Configuration "variable"	Value(s)
Number of racks	single, double
Network	1 Gbps
Nodes(total)	2, 4, 8, 16
CPU/node	2x Xeon Quad 2.5GHz
Disk/node	4x 750GB SATA

Table 1: Studied cluster configurations.

the simulation. For example, on receipt of a map task assignment message, the receiving node first needs to retrieve the associated input data. If the needed data is local, i.e., stored on the node itself, a disk read is simulated, otherwise a remote retrieval is simulated by sending a request message to a node that has the data and waiting for a response before proceeding to simulate the map function processing time. The process is repeated at each node, resulting in a desired simulated environment.

Finally, when handling network traffic, no real data is transferred in *MRPerf*. Rather size of data is used by ns-2 to calculate network traffic and transfer latencies.

3.4 Capturing Key-Value Distribution

In MapReduce, the distribution of key-value pairs between the map and reduce phases is application and data dependent. Thus, the distribution may become unbalanced. An unbalanced distribution results in some reduce tasks being assigned disproportionally large amount of work, thus causing them to take significantly longer to complete, compared to other tasks. Since the simulator does not process actual data, such distribution imbalance is not captured. We faced the decision to account for imbalance at the cost of complicating *MRPerf* design. On studying real applications, we observed that for most of the MapReduce jobs, there is only very small variance in distribution across nodes. Thus, in our current design, we assume uniform distribution of key-value pairs; each reduce task receives an equal part from each map task's output. However, our design is flexible and complex distributions can be added to *MRPerf*, if the need to do so arises.

In summary, *MRPerf* allows for realistically simulating MapReduce setups, and the design is flexible enough to capture a vast variety of configurations and job characteristics.

4. EVALUATION

We have implemented *MRPerf* using a mix of C++, tcl, and python code (3372 lines total) interfaced with the ns-2 simulator. In this section, we validate performance prediction made by *MRPerf* using performance results from a real-world application run on a medium-scale Hadoop [9] cluster. We present results of validation on a single-rack topology and a double-rack topology, validation at sub-phase level, detailed comparison of a single job, and look at jobs with different input size/chunk size. Next, we present two patches we made to Hadoop, in order to match performance prediction made by *MRPerf* to Hadoop. We note that our initial evaluation focus on *MRPerf*s ability to capture Hadoop behavior and result verification. Our benchmark application makes full use of the available resources, but does not overload them. Studying *MRPerf* under overload conditions, however, is a focus of our ongoing research.

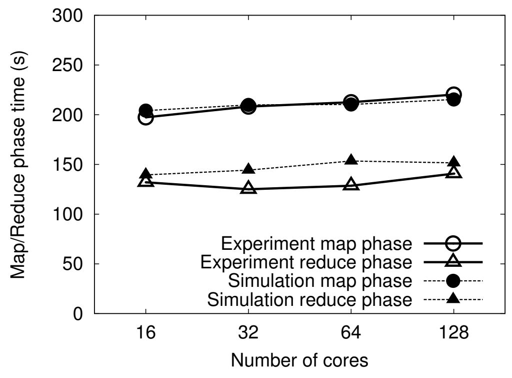

Figure 2: Execution times using actual measurements and *MRPerf* for single rack configuration.

4.1 Validation Tests

In the first set of experiments, we collected data from a number of real cluster configurations and compared it with that observed through *MRPerf*. Table 1 shows the cluster configurations studied for the validation tests. For our initial tests, we used a simple point-to-point connection when using multiple racks, however, this can be modified to more advanced topologies as needed.

For the validation tests, we used the TeraSort application as the benchmark. TeraSort [6] is designed for sorting terabytes of data. It samples the input data and uses map/reduce to sort the data into a total order. TeraSort is a standard map/reduce sort, except for a custom partitioner that uses a sorted list of $N-1$ sampled keys that define the key range for each reduce. In particular, all keys such that $sample[i-1] \leq key < sample[i]$ are sent to reduce i. This guarantees that the output of reduce i are all less than the output of reduce $i+1$.

We collect data by running TeraSort on a real Hadoop cluster with a chunk size of 64 MB and an input of 4GB/node (i.e. 64 GB input data for 16-node cluster), and then compare these results with those obtained through *MRPerf*.

4.1.1 Single Rack Cluster

In the first validation test, we utilize a number of compute nodes arranged in a single Hadoop rack. We vary the number of cores from 16 to 128 (2 to 16 nodes), and observe the total execution time for TeraSort. Figure 2 shows the results for the actual runs as well as numbers predicted by *MRPerf*. The break down for each case is shown in terms of map and reduce phases. The results show that *MRPerf* is able to predict the map phase performance within 3.42% of the measured values. The reduce phase simulated results are within 19.32% of the measured values. Overall, we see that *MRPerf* is able to predict Hadoop performance fairly accurately as we go from 16 to 128 cores.

4.1.2 Double Rack Cluster

Next, we repeated the above validation test with a two rack cluster, with racks connected to each other over 1Gbps link. Once again, we varied the total number of resources from 16 to 128 cores, with each rack containing half the resources. Figure 3 shows the results. Here, we once again observe a good match between simulated and actual measurements. The exception is the map phase performance for

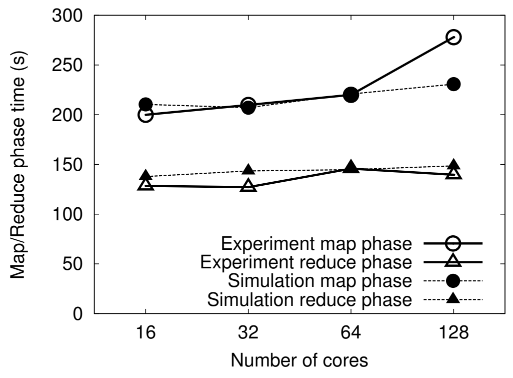

Figure 3: Execution times using actual measurements and *MRPerf* for double rack configuration.

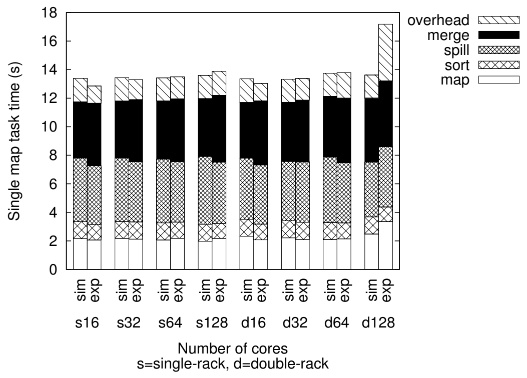

Figure 4: Sub-phase break-down times using actual measurements and *MRPerf*.

the 128-core case. Here, the predicted values are 16.99% lower than the actual processing time. On further investigation, we observed low network throughput on the inter-rack link and some network errors reported by the application, which we suspect are due to packet drops at the router in our experimental testbed (possibly due to the TCP incast [17]). The network slow-down caused the map phase taking longer than predicted since our model assumes a high-performance router connecting the two racks. We continue to develop means for better modeling such routers within ns-2, however, such router modeling is orthogonal to this work. Excluding the diverge of map phase in 128-core case, *MRPerf* is able to predict performance within 5.22% for the map phase and within 12.83% for the reduce phase, compared to the actual measurements.

4.2 Sub-phase Performance Comparison

So far, we have presented a comparison of overall execution times obtained via simulation and actual measurement. In the next experiment, we break a map task in further sub-phases, namely *map*, *sort*, *spill*, *merge*, and *overhead*. A *map* reads the input data, and processes it. The output is buffered in memory, and is sorted in memory during *sort*. The data is then written to the disk during *spill*. If multiple spills are involved, the data is read into memory once again for merging during *merge*. Finally, *overhead* accounts for miscellaneous processing outside of the above sub-phases, such as message passing via network. Figure 4 shows the sub-phase break-up times for 16 to 128 core cluster under *MRPerf* and actual measurements. Each cluster of bars labeled with a prefix of "s" stands for results from a single-rack topology, and a prefix of "d" stands for results from a double-rack topology. The following number is number of cores. As can be observed, *MRPerf* is able to provide very accurate predictions for performance, even at sub-phase level. Once again, we see that the network problem discussed above resulted in a larger overhead for 128-core case. However, other sub-phases are reasonably captured by *MRPerf*. The other simulated results are within error range of 13.55% compared to actual measurements.

Overview	Actual		*MRPerf*	
Number of map tasks	480		476	
Number of reduce tasks	16		16	
Total input data	32G		32G	
Total output data	32G		32G	
Phases	Actual		*MRPerf*	
Map	220.0		220.8	
Shuffle	7.4		5.4	
Sort	0.5		3.4	
Reduce	137.9		135.9	
Map break-down	Actual		*MRPerf*	
map	2.14		2.10	
sort	1.12		1.19	
spill	4.22		4.58	
merge	4.52		4.26	
overhead	1.79		1.61	
sum	13.80		13.75	
Data locality	Actual		*MRPerf*	
	num	time	num	time
Data-local	468	13.77	468	13.66
Rack-local	6	13.60	3	14.67
Rack-remote	6	16.10	5	21.64

Table 2: Detailed characteristics of a TeraSort job.

4.3 Detailed Single-Job Comparison

In the next experiment, we focus on a single job and present a detailed comparison of the job's performance and workload under actual measurements and *MRPerf*. Table 2 shows the results. The selected job runs on 64 cores divided into 2 racks. Total input data size is 32 GB. The first part of the table is the overview of the TeraSort instance used for this test. The difference in the number of map tasks is due to the different way the input data is generated. For the actual run, the input is generated in a distributed manner by another application TeraGen, whereas in the simulator, input is generated randomly by data layout generator. Our generator always produces as many full chunks as possible, but since TeraGen works in a distributed manner, a few chunks created by it are not full-size. The second part of the table shows the total time of the MapReduce phases, as already seen in Figure 3 and Figure 4. The last part of the table shows the average performance of map tasks in different categories. Data-local map tasks are tasks that process data located on the same node on which a task is running. Rack-local map tasks are tasks that process data located in the same rack. Finally, rack-remote map tasks are tasks that

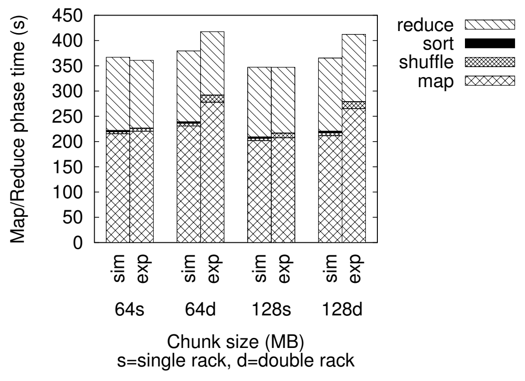

Figure 5: Execution times with varying chunk size using actual measurements and *MRPerf*.

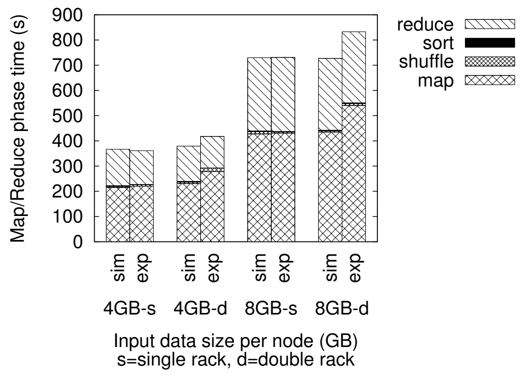

Figure 6: Execution times with varying input size using actual measurements and *MRPerf*.

process data located in another rack. For the presented job, most map tasks are data-local, and simulation shows similar performance for these tasks as observed through the experiments. The simulation also produces similar mix of three categories of map tasks. Overall, even at this granularity, the simulated results are quite similar to the actual results.

4.4 Validation with Varying Input

We have so far considered various topologies and number of nodes, but have used the same input size of 4 GB per node and a chunk size of 64 MB. Next, we fix the number of cores to 128, and study the 64 MB as well as 128 MB chunk size both under a single rack and double rack configuration. Figure 5 shows the results. We also study input data size of 4GB per node vs. 8GB per node under a single rack and double rack configuration. Figure 6 shows results for different input data size. These results show that *MRPerf* is able to correctly predict performance even for varying input and chunk sizes, and illustrates the simulators capabilities in capturing Hadoop cluster behavior.

4.5 Hadoop Improvements

While comparing application performance as predicted by *MRPerf* and real application performance with Hadoop we found several places where Hadoop didn't perform as well as predicted. In some cases we had to tweak our simulator to more closely model the Hadoop implementation but in other

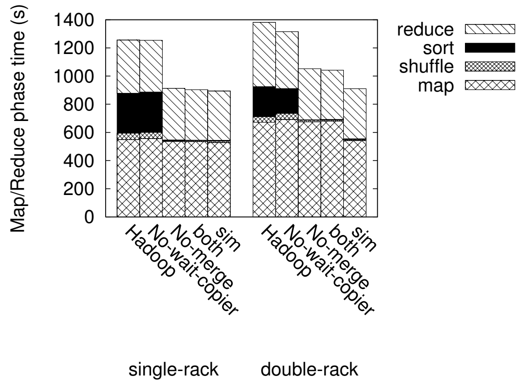

Figure 7: Performance improvement in Hadoop as a result of fixing two bottlenecks.

cases we found that Hadoop was making sub-optimal choices that decreased performance. In this section, we discuss two improvements we made to Hadoop based on predictions obtained from *MRPerf*.

By default, during the reduce phase, Hadoop merge-sorts 10 files at a time. We found this to be inefficient for our application and configurations and created a patch, *no-merge*, which does not perform file merges at shuffle time. The effect is similar to setting Hadoop's `io.sort.factor` parameter to a large value (but the value would need to be determined before the application is run.) However, this optimization does not come for free. To merge more files in one pass, more memory is needed. If total amount of memory is fixed, then each file would get a smaller buffer, and as disk seek time cannot be amortized by the shorter I/Os, the disk I/O performance would drop. That is why the reduce sub-phase in patched Hadoop exhibits a slow down, as seen in Figure 7. We have learned that Hadoop developers are aware of the problem and the trade-off between memory and disk I/O performance [5].

Another anomaly that we observed was that the network bandwidth during the shuffle phase in the experimental setup was not as high as predicted by *MRPerf*. We found that this was because the Hadoop framework did not have enough copy threads pulling data in parallel over the network. The framework launches copiers at most every second. If copiers finish quickly, new ones are not launched until the next second, which wastes available network bandwidth. We implemented a patch, *no-wait-copier*, that breaks up a single thread in Hadoop used normally for handling fetching notifications, launching copiers, and updating finished copiers, into separate threads dedicated to each of these tasks. With this patch, copiers are launched right after previous copiers finish, and bandwidth is utilized efficiently.

We applied both patches to Hadoop, and found that the patched version of Hadoop runs faster and matches the performance predicted by *MRPerf*. Figure 7 compares the performance of patched and unpatched versions of Hadoop with results from *MRPerf*. The application is the same as before, with 10GB input data per node and 64MB chunk size. Single-rack experiments are run on 128 cores (16 nodes) in 1 rack, and double-rack experiments are run on 128 cores (16 nodes) in 2 racks (8 in each). Single-rack results show *no-merge* has a huge improvement over Hadoop. Double-rack results also show effectiveness of the *no-merge*, as well as show that *no-wait-copier* improves performance only a little. Hadoop with both patches can save 28.05% and 17.02% over Hadoop in single-rack and double-rack cases, respectively. Predictions from *MRPerf* best match performance of Hadoop with both the patches. In double-rack case, the difference in map phase times can be explained by the network problems described above. Other than that, errors in predicted values for map/reduce phases are within 1.79% of the actual measurements.

5. RELATED WORK

MapReduce is an emerging model, and simulation-based approach to capture its performance has not been studied previously. However, a closely related large-scale distributed computing paradigm is Grid computing [16]. Grid computing is a well-established paradigm and has been used to solve large-scale problems using distributed resources. It addresses similar issues as MapReduce, but with a grander scope. A variety of simulators have been developed to model and simulate the performance of Grid systems including Bricks [8], Microgrid [19], Simgrid [12], and GridSim [11]. The interest in using simulation to model distributed systems can be gauged from the fact that the SourceForge project for GridSim shows over 5000 downloads between September 2007 and March 2009. In contrast to these simulators, *MRPerf* is focused on modeling the specifics of MapReduce frameworks and not grid systems, so it does not worry about reservations and wide-area scheduling decisions that are critical for Grids.

The desire to understand the performance of MapReduce systems has led to a variety of efforts, including the Chukwa project [10] and instrumenting Hadoop using X-Trace [15]. These efforts are complimentary to our work, and in the future we hope to modify *MRPerf* to produce output reports in the same formats as these systems, so that analysis tools developed for these systems can also be used to study results from *MRPerf*.

6. CONCLUSION

In this paper, we have presented the design and evaluation of *MRPerf*, a phase-level simulator for the widespread MapReduce model, to better design, provision, and fine-tune Hadoop systems. *MRPerf* enables realistic simulations for analyzing performance of applications on specific Hadoop configurations, and can be employed as a planning tool to evaluate proposed cluster designs and topologies. Several effective simplifying assumptions are made in the design of *MRPerf*, which allow it to quickly yet accurately model application behavior. Our validation of *MRPerf* using small- to medium-scale clusters shows that it can realistically simulate Hadoop's behavior. Moreover, our experience during the development of the simulator lead us to uncover performance issues with the Hadoop implementation, fixing which resulted in a performance improvement of up to 28.05% for a typical sort application. Thus, *MRPerf* provides a promising tool to system designers and application developers, which can be used to customize MapReduce environments.

Acknowledgment

This research is supported in part by NSF (CAREER Award CCF-0746832), and IBM through IBM Faculty Award (Virginia Tech Foundation grant VTF-874574).

7. REFERENCES

[1] DiskSim, Aug 2008. `http://www.pdl.cmu.edu/DiskSim/`.

[2] ns-2, Aug 2008. `http://nsnam.isi.edu/nsnam/index.php/Main_Page`.

[3] Disco Project, Jan. 2009. `http://discoproject.org/`.

[4] Hadoop User Mailing List Archive, Mar. 2009. `http://mail-archives.apache.org/mod_mbox/hadoop-core-user/`.

[5] JIRA: HADOOP-3473, Feb 2009. `http://issues.apache.org/jira/browse/HADOOP-3473`.

[6] Terasort, Mar 2009. `http://hadoop.apache.org/core/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html`.

[7] Adam Pisoni. Skynet, Apr. 2008. `http://skynet.rubyforge.org`.

[8] K. Aida, A. Takefusa, H. Nakada, S. Matsuoka, S. Sekiguchi, and U. Nagashima. Performance Evaluation Model for Scheduling in Global Computing Systems. *Int. J. High Perform. Comput. Appl.*, 14(3):268–279, 2000.

[9] Apache Software Foundation. Hadoop, May 2007. `http://hadoop.apache.org/core/`.

[10] J. Boulon, A. Konwinski, R. Qi, A. Rabkin, E. Yang, and M. Yang. Chukwa, a large-scale monitoring system. In *Proc. CCA*, 2008.

[11] R. Buyya and M. M. Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *CoRR*, cs.DC/0203019, 2002.

[12] H. Casanova. Simgrid: A Toolkit for the Simulation of Application Scheduling. In *Proc. IEEE CCGRID*, 2001.

[13] J. Dean. Experiences with mapreduce, an abstraction for large-scale computation. In *Proc. IEEE PACT*, 2006.

[14] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Comm. of the ACM*, 51(1):107–113, 2008.

[15] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *Proc. USENIX NSDI*, 2007.

[16] I. Foster (Ed.) and C. Kesselman (Ed.). *The GRID: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, 1999.

[17] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *Proc. USENIX FAST*, 2008.

[18] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proc. IEEE HPCA*, pages 13–24, 2007.

[19] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien. The MicroGrid: A scientific tool for modeling Computational Grids. *Sci. Program.*, 8(3):127–141, 2000.