A Simulation Approach to Evaluating Design Decisions in MapReduce Setups

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

Prashant Pandey, Karan Gupta
IBM Almaden Research Center
San Jose, CA
Email: {ppandey,guptaka}@us.ibm.com

***Abstract*—MapReduce has emerged as a model of choice for supporting modern data-intensive applications. The model is easy-to-use and promising in reducing time-to-solution. It is also a key enabler for cloud computing, which provides transparent and flexible access to a large number of compute, storage and networking resources.**

Setting up and operating a large MapReduce cluster entails careful evaluation of various design choices and run-time parameters to achieve high efficiency. However, this design space has not been explored in detail. In this paper, we adopt a simulation approach to systematically understanding the performance of MapReduce setups. The resulting simulator, *MRPerf*, captures such aspects of these setups as node, rack and network configurations, disk parameters and performance, data layout and application I/O characteristics, among others, and uses this information to predict expected application performance.

Specifically, we use *MRPerf* to explore the effect of several component inter-connect topologies, data locality, and software and hardware failures on overall application performance. *MRPerf* allows us to quantify the effect of these factors, and thus can serve as a tool for optimizing existing MapReduce setups as well as designing new ones.

I. Introduction

Cloud computing is emerging as a viable model for enabling fast time-to-solution for modern large-scale data-intensive applications. The benefits of this model include efficient resource utilization, improved performance, and ease-of-use via automatic resource scheduling, allocation, and data management. Increasingly, the MapReduce [1] framework is employed for realizing cloud computing infrastructures, which simplifies the application development process for highly-scalable computing infrastructures. Designing 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. With the exception of some site-specific insights, e.g., Google's MapReduce infrastructure [2], this design space is mostly unexplored. However, estimating how applications would perform on specific MapReduce setups is critical, especially for optimizing existing setups and building new ones.

In this paper, we adopt a simulation approach to explore the impact of design choices in MapReduce setups. We are concerned with how decisions about cluster design, run-time parameters, multi-tenancy and application design affect application performance. We develop an accurate simulator, *MRPerf*, to comprehensively capture the various design parameters of a MapReduce setup. *MRPerf* can help quantify the affect of various factors on application performance, as well as capture the complex interactions between the factors. We expect *MRPerf* to be used by researchers and practitioners to understand how their MapReduce applications will behave on a particular setup, and how they can optimize their applications and platforms. The overarching goal is to facilitate MapReduce deployment via use of *MRPerf* as a feedback tool that provides systematic parameter tuning, instead of the extant inexact trial-and-error approach.

Current trends show that MapReduce is considered a high-productivity alternative to traditional parallel programming paradigms for enterprise computing [2], [3], [4] as well as scientific computing [5], [6]. Although MapReduce, especially its Hadoop [3] implementation, is widely used, its performance for specific configurations and applications is not well understood. In fact, a quick survey of related discussion forums [7] reveals that most users are relying on rules-of-thumb and inexact 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. However, to achieve optimum system design, the scale and complexity of MapReduce setups create a deluge of parameters that require tuning, testing, and evaluating for optimum system design. *MRPerf* aims to answer questions being asked by the community about MapReduce setups: How well does MapReduce scale as the cluster size grows large, e.g., 10,000-nodes? Can a particular cluster setup yield a desired I/O throughput? Can a MapReduce application provide linear speed-ups as number of machines increases? Moreover, *MRPerf* can be used to understand the sensitivity of application performance to platform parameters, network topology, node resources and failure rates.

Building a simulator for MapReduce is challenging. First, choosing the right level of component abstraction is an issue: If every component is simulated thoroughly, it will take prohibitively long to produce results; conversely, if important components are not thoroughly modeled, results may lack desired accuracy and detail. Second, 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. Third, the shuffle/sort and reduce phases of a MapReduce application are dependent on the input and require special consideration for correct simulations. Fourth, correctly modeling failures is critical, as failures are common in large scale commodity clusters and directly affect performance. Finally, verifying *MRPerf* at scale is complex as it 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 *MRPerf* is to take on these challenges and answer the above questions, as well as explore the impact of factors such as data-locality, network topology, and failures on overall performance.

A. Our Contributions

In this paper, we make the following contributions:

- Design, develop, and implement an accurate Hadoop [3] simulator, *MRPerf*, which models execution on specified MapReduce setups, and validate it using measurements on actual Hadoop setups;
- Apply *MRPerf* to study the impacts of different network topologies on Hadoop application performance;
- Investigate and quantify the role of data locality on application performance; and
- Study how various failures affect Hadoop setup.

We have successfully verified *MRPerf* using a medium-scale (40-node) cluster. Moreover, we used *MRPerf* to quantify the impact of data-locality, network topology, and failures using representative MapReduce applications running on a 72-node simulated Hadoop setup, and gained key insights. For example, for the *TeraSort* [8] application, we found that: advanced cluster topologies, such as *DCell* [9], can improve performance upto 99% compared to a common *Double rack* topology; data locality is crucial to extracting peak performance with a node-local task placement performing 284% better than rack-remote placement in the *Double rack* topology; and MapReduce can tolerate failures in individual tasks with small impact, while network partitioning can reduce the performance by 60%.

II. Modeling Design Space

We are faced with modeling the complex interactions of a large number of factors, which dictate how an application will perform on a given MapReduce setup. These factors can be classified into design choices concerning infrastructure implementation, application management configuration, and framework management techniques. A summary of key design parameters modeled in *MRPerf* is shown in Table I.

MapReduce infrastructures typically encompass a large number of machines. A *rack* refers to a collection of compute nodes with local storage. It is often installed on a separate machine-room rack, but can also be a logical subset of nodes. Nodes in a rack are usually a single network hop away from each other. Multiple racks are connected to each other using a hierarchy of switches to create the cluster. Thus, the infrastructure design parameters involve varying node capabilities and interconnect topologies. In, *MRPerf*, we categorize these critical parameters as *cluster parameters*, and they can have a profound impact on overall system performance.

TABLE I
MapReduce setup parameters modeled in *MRPerf*.

Category	Example
Cluster parameters	• Node CPU, RAM, and disk charactersitics • Node & Rack heterogeneity • Network topology (inter & intra-rack)
Configuration parameters	• Data replication factor • Data chunk size used by the storage layer • Map and reduce task slots per node • Number of reduce tasks in a job
Framework parameters	• Data placement algorithm • Task scheduling algorithm • Shuffle-phase data movement protocol.

The ease-of-use of the MapReduce programming model comes from its ability to automatically parallelize applications — most MapReduce applications are embarrassingly parallel in nature — to run across a large number of resources. Simply put, MapReduces splits an application's input dataset into multiple tasks and then automatically schedules these tasks to available resources. The exact manner in which a job's data gets split, and when and on what resources the resulting tasks are executed, is influenced by a variety of *configuration parameters*, and is an important determinant of performance. These parameters capture inherent design trade-offs. For example: Splitting data into large chunks yields better I/O performance (due to larger sequential accesses), but reduces the opportunity for running more parallel tasks that are possible with smaller chunks; Replicating the data across multiple racks provides easier task scheduling and better data locality, but increases the cost of data writes (requiring updating multiple copies) and slows down initial data setup.

Finally, design and implementation choices within a MapReduce framework also affect application performance. These *framework parameters* capture setup management techniques, such as how data is placed across resources, how tasks are scheduled, and how data is transferred between resources or task phases. These parameters are inter-related. For instance, an efficient data placement algorithm would make it easy to schedule tasks and exploit data locality.

The job of *MRPerf* is further complicated by the fact that the impact of a specific factor on application behavior is not constant in all stages of execution. For example, the network bandwidth between nodes is not an important factor for a job that produces little intermediate output if the map tasks are scheduled on nodes that hold the input data. However, 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. *MRPerf* should model these interactions to correctly capture the performance of a given MapReduce setup.

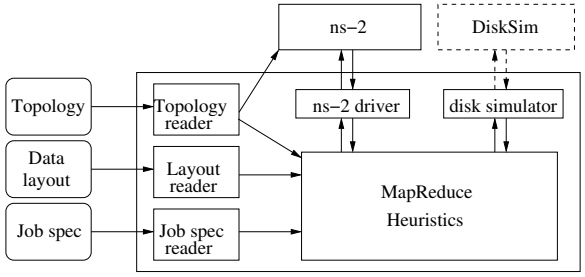


Fig. 1. *MRPerf* architecture.

III. Design

In this section, we present the design of *MRPerf*. Our prototype is based on Hadoop [3], the most widely-used open-source implementation of the MapReduce framework.

A. Architecture Overview

The goal of *MRPerf* is to provide fine-grained simulation of MapReduce setups at sub-phase level. On one hand, it models inter- and intra-rack interactions over the network, on the other hand, it models single node processes such as task processing and data access I/O time. Given the need for accurately modeling network behavior, we have based *MRPerf* on the well-established ns-2 [10] network simulator. The design of *MRPerf* is flexible, and allows for capturing a wide-variety of Hadoop setups. To use the simulator, one has to provide node specification, cluster topology, data layout, and job description [11]. The output is a detailed phase-level execution trace that provides job execution time, amount of data transferred, and time-line of each phase of the task. The output trace can also be visualized for analysis.

Figure 1 shows the high-level 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 provides the interface for network simulation. Similarly, the disk module provides modeling for the disk I/O. Although we use a simple disk model in this study, the disk module can be extended to include advanced disk simulators such as DiskSim [12]. All the modules are driven by the MapReduce Heuristics module (MRH) that simulates Hadoop's behavior. To perform a simulation, *MRPerf* first reads all the configuration parameters and instantiates the required number of simulated nodes arranged in the specified topology. The MRH then schedules tasks to the nodes based on the specified scheduling algorithm. This results in each node *running* its assigned job, which further creates network traffic (modeled through ns-2) as nodes interact with each other. Thus, a simulated MapReduce setup is created.

We make two simplifying assumptions in *MRPerf*. (i) A node's resources, i.e., processors and disks, are equally shared among tasks assigned concurrently to the node. (ii) *MRPerf* does not model OS-level asynchronous prefetching. Thus, it only overlaps I/O and computation across threads and processors (and not in a single thread). These assumptions may cause some loss in accuracy, but greatly improve overall simulator design and performance.

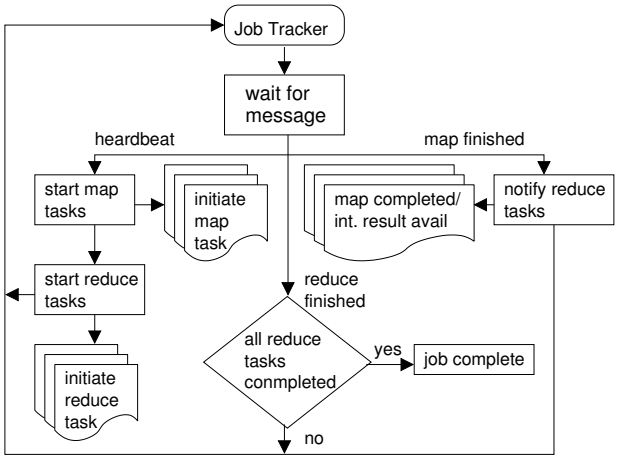


Fig. 2. Control flow in the Job Tracker.

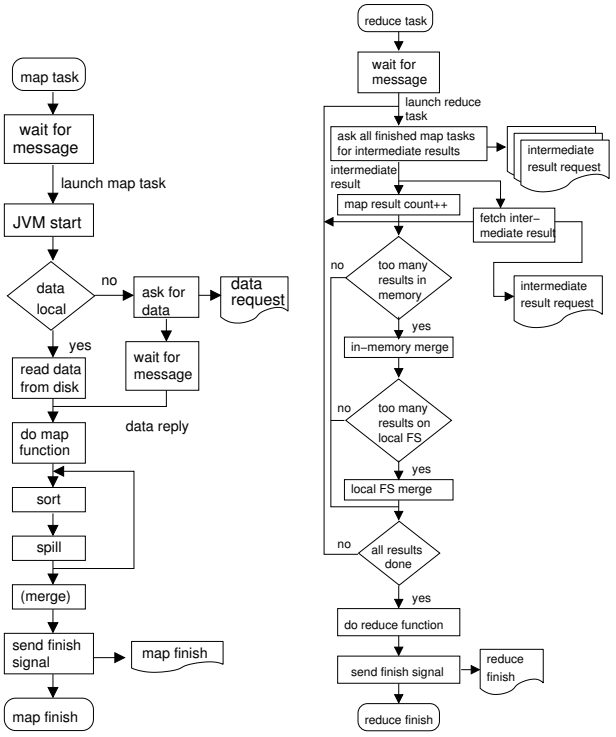
B. Simulating Map and Reduce Tasks

MRPerf employs packet-level simulation and relies on ns-2 for capturing network behavior. The main job of *MRPerf* is to simulate the map and reduce tasks, manage their associated input and output data, make scheduling decisions, and model disk and processor load. To model a setup, *MRPerf* creates a number of simulated nodes. Each node has several processors and a single disk, and the processing power is divided equally between the jobs scheduled for the node. Also, each simulated node is responsible for tracking its own processor and disk usage, and other statistics, which is periodically written to an output file.

Our design makes extensive use of the `TcpApp Agent` code in ns-2 to create functions that are triggered (called-back) in response to various events, e.g., receiving a network packet. *MRPerf* utilizes four different kinds of agents, which we discuss next. Note that a node can run multiple agents at the same time, e.g., run a map task and also serve data for other nodes. Each agent is a separate thread of execution, and does not interfere with others (besides sharing resources).

a) Tracking job progress: The main driver for the simulator is a Job Tracker that is responsible for spawning map and reduce tasks, keeping a tab on when different phases complete, and producing the final results. Figure 2 shows the control flow diagram for the Job Tracker. Most of the behavior is modeled in response to receiving messages from other nodes. However, the Job Tracker also has to perform tasks, such as starting new map and reduce operations as well as bookkeeping, which are not in response to explicit interaction messages. *MRPerf* uses a heartbeat trigger to initiate such Job Tracker functions, and to capture the correct MapReduce behavior.

b) Modeling map task: Receipt of a message from the Job Tracker to start a map task results in the sequence of events shown in Figure 3(a). (i) A Java VM is instantiated for the task. (ii) Necessary data is either read from the local



(a) Map task (b) Reduce task

Fig. 3. Control flow when running map and reduce tasks.

disk or requested remotely. If a remote read is necessary, a data request message is sent to the node that has the data, and the process stalls until a reply with the data is received. (iii) Application-specific map, sort, and spill operations are performed on the input data until all of it has been consumed. (iv) A merge operation, if necessary, is performed on the output data. Finally, (v) a message indicating the completion of the map task is returned to the Job Tracker. The process then waits for the next assignment from the Job Tracker.

c) Modeling reduce task: The reduce task is also initiated upon receiving a message from the Job Tracker. The sequence of events in this task, as shown in Figure 3(b), are as follows. (i) A message is sent to all the corresponding map tasks to request intermediate data. (ii) Intermediate data is processed as it is received from the various map tasks. If the amount of data exceeds a pre-specified threshold, an in-memory or local file system merge is performed on the data. These two steps are repeated until all the associated map tasks finish, and the intermediate data has been received by the reduce task. (iii) The application-specific reduce function is performed on the combined intermediate data. Finally, (iv) similarly as for the map task, a message indicating the completion of the reduce task is sent to the Job Tracker, and the process waits for its next assignment.

d) Simulating data access: Another critical task in *MRPerf* is properly modeling how data is accessed on a node. This is achieved through a separate process on each simulated node, which we refer to as the Data Manager. Briefly, the main job of the Manager is to read data (input or intermediate) from the local disk in response to a data request, and send the requested items back to the requester. Separating data access from other tasks has two advantages. First, it models the network overhead of accessing a remote node. Second, it provides for extending the current disk model with more advanced simulators, e.g., DiskSim [12].

Finally, to reduce simulation overhead, we do not perform packet-level simulations for the actual data, which is done only for the meta-data. Instead, we use the size of the data and the bandwidth observed through ns-2 to calculate transfer times for calculating overall task execution times.

C. Input Specification

The user input needed by *MRPerf* 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.

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 operations, 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-phases 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.

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 in *MRPerf*.

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

D. Limitations of our Simulator

The current implementation of *MRPerf* is limited to modeling a single storage device per node, supporting only one replica for each chunk of output data (input data replication is supported), and not modeling certain optimizations such as speculative execution. We support simple node and link failures, but more advanced exceptions, such as a node running slower than others or partially failing, are not currently

modeled. However, we stress that lack of such support does not restrict *MRPerf*'s ability to model performance of most Hadoop setups. Nonetheless, since such support will enhance the value of *MRPerf* and enable us to investigate Hadoop setups more thoroughly, addressing these limitations is the focus of our ongoing research.

In summary, *MRPerf* allows for realistically simulating MapReduce setups, and its design is extensible and flexible. Thus, *MRPerf* can capture a wide-range of configurations and job characteristics, as well as evolve with newer versions of Hadoop.

IV. Evaluation

There are four goals of our evaluation: (i) validate *MRPerf*'s performance predictions using real-world applications running on a medium-scale Hadoop [3] cluster; (ii) use *MRPerf* to study the role of network topology on the application performance; (iii) use *MRPerf* to demonstrate the importance and impact of data locality for different MapReduce applications; and (iv) apply *MRPerf* to study the impact of infrastructure failures. For this purpose, we have implemented *MRPerf*, as described in Section III using about 4000 lines of C++, Tcl, and Python code interfaced with the ns-2 simulator.

A. Applications

We have used several representative MapReduce applications in our evaluation study. In the following, we present a brief description of these applications.

- *TeraSort.* The *TeraSort* application [13] is motivated by the TeraSort benchmark [8], which measures the time needed to sort 10 billion 100 byte records. Sorting is an important step in many analytics applications and stresses the infrastructure by producing as much intermediate data (which needs to be shuffled) and final output (which needs to be saved in the distributed filesystem) as the input.
- *Search.* In this synthetic application, we model a search application that compares each input record with a set of match criteria, and finds a small subset of matches. The complexity of match criteria determines the CPU load of the map tasks. *Search*, parameterized by match-complexity, allows us to study the impact of varying map times with fixed input and output size.
- *Index.* In this synthetic application, we model an indexing application that generates map (and reduce) output for each unique word found in the input data. The amount of output data depends on the number of unique words in the input data. *Index*, parameterized by the fraction of unique input words, allows us to study the impact of varying intermediate data size (map-side output) with fixed map times.

Table II summarizes the application variants used in the study and the corresponding value of the key parameters: `Cycles/byte` represents the compute cycles spent per input byte and captures the compute-complexity of the application; `Filter Percentage` captures the ratio between the size of input and output data during the map phase, and is specified using a minimum and maximum value. For instance, *TeraSort* spends 40 cycles per input byte, on average, and the output size is equal to the input size. Whereas, *Search(c)* spends 400 cycles per input byte, and the output can range from size zero, i.e., searched term not found, to a small fraction (0.01%) of the input size.

TABLE II
Parameters of the synthetic applications used in the study.

App	Cycles/byte	Min. Filter %	Max. Filter %
TeraSort	40	100%	100%
Search(a)	4	0%	0.01%
Search(b)	40	0%	0.01%
Search(c)	400	0%	0.01%
Index(a)	40	2%	2%
Index(b)	40	10%	10%
Index(c)	40	50%	50%

TABLE III
Studied cluster configurations.

Configuration parameters	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

B. Validating MRPerf Design

In this section, we present a brief validation of the performance predictions made by *MRPerf* using measurements of a real-world application run on a medium-scale Hadoop cluster. Detailed validation results can be found in our earlier work [11]. For the presented results, we measured the performance of *TeraSort* (chunk size 64 MB, input 4GB/node) on a cluster configurations shown in Table III, and compared the results with those estimated by *MRPerf*.

1) Single-rack cluster test: In the first test, we use a single Hadoop rack containing all the nodes and one router. All node-router links are 1 Gbps. The number of nodes is varied (16 to 128 cores), and we observe the real and simulated total execution time for *TeraSort*. The results are shown in Figure 4, which shows the breakdown in terms of map and reduce phases. Here, we observe that *MRPerf* is able to predict the map and reduce phases performance within a range of 3.42% and 19.32% of the measured values, respectively. Moreover, *MRPerf* is able to predict Hadoop performance fairly accurately as the number of cores is increased from 16 to 128. Note that the average map time remains roughly the same across configurations because in all cases a single map task is processing the same amount of data (a single 64 MB chunk). The overall execution time and average reduce time also remain similar across configurations since the amount of data (and the number of reduce tasks) is scaled proportionally with cluster size.

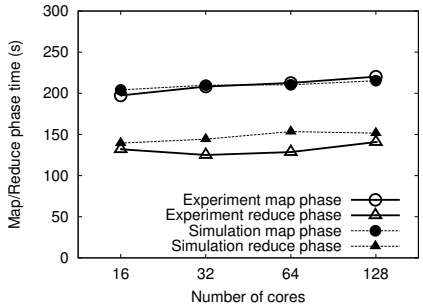


Fig. 4. *TeraSort* single-rack actual measurements vs. *MRPerf* execution times.

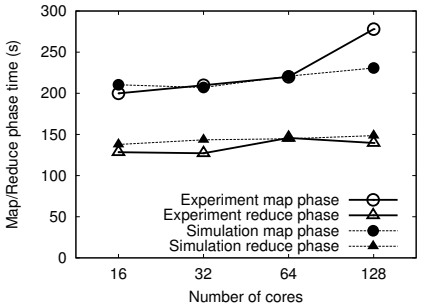


Fig. 5. *TeraSort* double-rack actual measurements vs. *MRPerf* execution times.

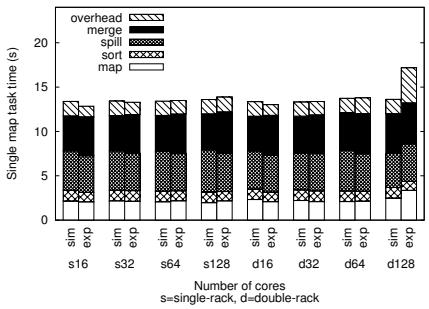


Fig. 6. *TeraSort* sub-phase break-down times using actual measurements and *MRPerf*.

2) Double rack cluster test: Next, we repeat the validation test with a cluster comprising two racks, with each rack containing half the nodes from before and one router, and the routers in the two racks connected using a single link. All node-router and router-router links are 1 Gbps. Figure 5 again shows a good match between simulated and actual measurements. The only exception is the map phase performance for the 128-core case, where the predicted values are 16.99% lower than the actual measured time. Further investigation revealed that the network throughput on the inter-rack link was much lower than expected and the application reported some network errors. One suspected reason for this is packet drops at the router in our testbed (possibly due to the TCP incast [14]). Since this unexpected network slow-down is not modeled in *MRPerf* as it assumes a high-performance router connecting the two racks, the increase in the map phase execution time was not predicted. We continue to develop means for better modeling of such routers in *MRPerf* using ns-2, however, better router modeling is orthogonal to this work. Excluding the 128-core case, *MRPerf* is able to predict performance within a range of 5.22% and 12.83% of the actual measurements for the map and reduce phases, respectively.

3) Comparing sub-phase performance: In the next experiment, we study the sub-phases of a map task. *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 network messages. Figure 6 shows the sub-phase break-up times for 16 to 128 core clusters under *MRPerf* and actual measurements, with prefixes "s" and "d" showing results for single-rack and double-rack topology, respectively. As evident, *MRPerf* is able to accurately predict performance even at sub-phase level. Once again, the router issue discussed earlier resulted in a larger overhead for the 128-core case. However, other sub-phases are reasonably captured by *MRPerf*. Overall, *MRPerf*'s results are within a range of 13.55% of the actual measurements.

In summary, this set of experiments show that *MRPerf* is capable of accurately simulating Hadoop behavior.

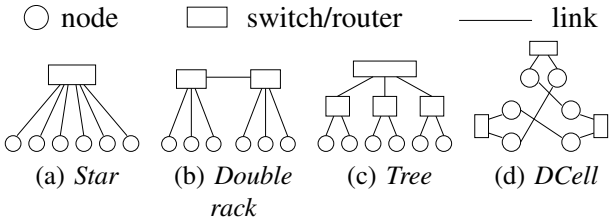


Fig. 7. Network topologies considered in this study. An example setup with 6 nodes is shown.

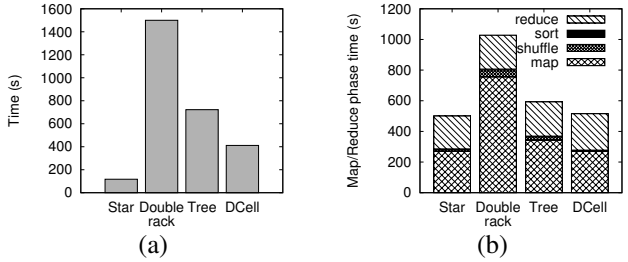


Fig. 8. Performance under studied topologies. (a) All-to-all messaging microbenchmark. (b) *TeraSort*.

C. Impact of Network Topology

In the next set of experiments, we utilize *MRPerf* to investigate the impact of network topology on Hadoop application performance. For this purpose, we simulate 72 nodes connected via 1 Gbps links. We consider four topologies as shown in Figure 7: *Star* where a single router connects all the 72 nodes; *Double rack* where the resources are divided equally into two racks of 36 nodes, each rack has its own router to connect all of its nodes, and the racks are connected using a point-to-point link between their routers; *Tree* where nodes are divided into 9 racks with 8 nodes each, and the racks are connected via a hierarchy of routers; and *DCell* [9], an advanced network topology, where nodes are distributed similarly as in *Tree* but the interconnectivity is recursively defined, with the nodes participating in the routing. The main advantage of *DCell* is that it does not require expensive switches with a large number of ports, rather cost-effective 8-port switches can be used to build large-scale setups.

1) Micro-benchmark test: We first evaluate the relative performance of the topologies using an all-to-all commu-

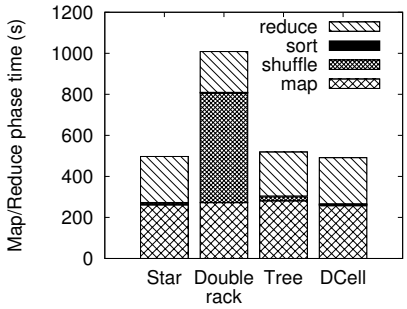


Fig. 9. *TeraSort* performance under studied topologies with all data available locally.

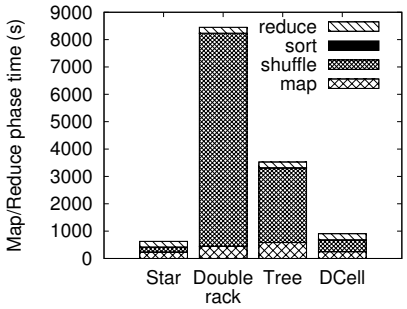


Fig. 10. *TeraSort* performance under studied topologies with all data available locally and 100 Mbps links.

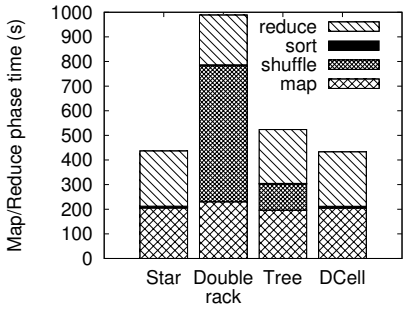


Fig. 11. *TeraSort* performance under studied topologies with all data available locally and using faster map tasks.

nication micro-benchmark (not a Hadoop job). In this test, each pair of nodes exchange data, with each node sending 1 MB of data to every other node, and repeating for 100 times. This experiment demonstrates the wide variation in total bandwidth in the different topologies in the presence of all-to-all communication. In particular, the hierarchical schemes (*Tree* and *Double rack*) end up with the higher-level links as bottlenecks in such communication. Figure 8(a) shows the total time for the 100 rounds of communication: *Star* has the best performance as links are not shared between nodes. *Double rack* and *Tree* bottleneck on link capacity, and thus perform a factor of 12x and 6x slower than *Star*, respectively. Finally, *DCell* is a factor of 3.5x slower compared to *Star* as inter-rack links are shared, but is a factor of 3.6x and 1.8x faster than *Double rack* and *Tree*, respectively. We repeated the experiment varying the number of rounds and the message sizes and obtained similar results. Based on these observations, we infer that *DCell* is a promising topology for use in Hadoop setups.

2) Effect on TeraSort: In our next experiment, we run *TeraSort* on each of the topologies in *MRPerf*. For each case, one node acts as the Job Tracker, while the remaining nodes run map and reduce tasks, as well as serve data. Figure 8(b) shows the results. *DCell* is able to perform as well as *Star* since the network usage is slowed down (compared with the previous experiment) by sorting being done by the nodes. *Double rack* and *Tree* are slower, taking 99% and 15% more time, respectively, compared to *Star*. The map phase in the different topologies is not identical because some map tasks retrieve their input data over the network, which takes a longer time when the network is overloaded.

a) Eliminating the effect of remote data retrieval: In the next experiment, we modify several settings to isolate the impact of network topology on the reduce phase. We modify the previous experiment in three ways: (A) make all data for the reduce phase available locally; (B) change all links to be 100 Mbps instead of 1 Gbps; and (C) use faster map tasks. Figure 9 shows the performance of the topologies with A. As expected, the map times are similar for all the four topologies. Moreover, since the amount of data transferred over the network is reduced, the network is less

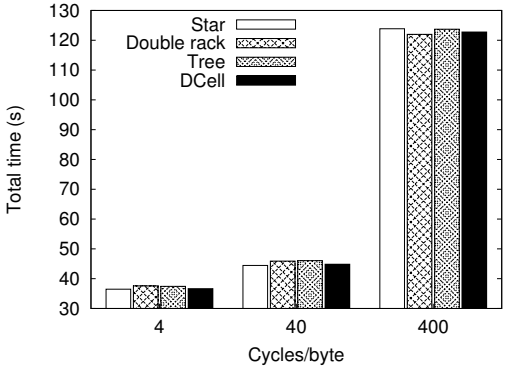


Fig. 12. *Search* performance under studied topologies with 100 Mbps links.

of a bottleneck and the shuffle, sort, and reduce performance is also similar. The only exception is *Double rack*, where the shuffle phase takes a long time, since the single link between the two racks is still a bottleneck in the all-to-all data transfer needed during shuffle. To highlight the effect of topologies, we repeat the experiment with slower links and local map data, i.e., with both A and B. Figure 10 shows the results for this case. Now, the network becomes a bottleneck for *Tree*, *Double rack* and *DCell* during the shuffle phase, but *DCell* is the closest to *Star* due to its higher aggregate bandwidth for all-to-all communication. Finally, we modified the setup to use 20% faster map tasks with local map data, i.e., with A and C. The motivation is to increase the rate of data produced for shuffling, and thus to highlight any network bottleneck if present. Figure 11 shows a similar behavior as before, illustrating that even for medium-sized clusters and 1 Gbps networks, inter-node bandwidth can be the bottleneck for MapReduce applications.

3) Effect on Search: Next we study the affect of network topologies on *Search*. Again, the simulations model a 72-node topologies, 1 GB input data per node, and a 64 MB block size (with map input data available locally). *Search* produces an insignificant amount of intermediate data and thus is largely unaffected by network topologies. Figure 12 shows that the application's performance is fairly even across all topologies (with 100 Mbps links). Experiments with 1 Gpbs

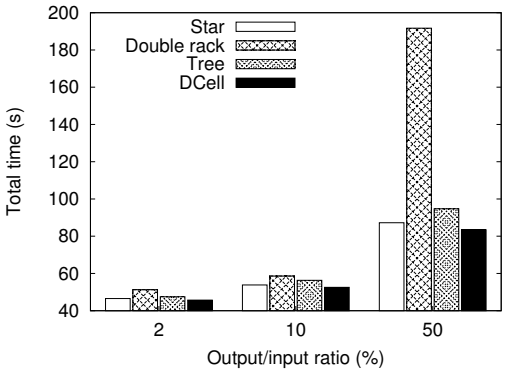


Fig. 13. *Index* performance under studied topologies.

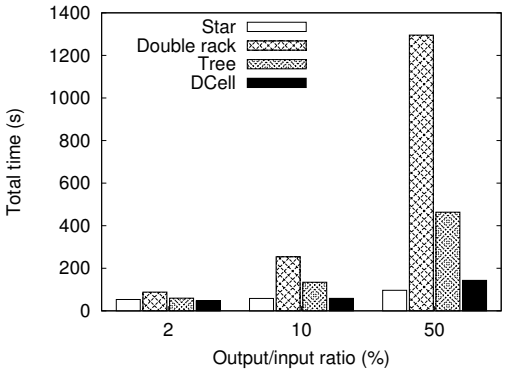


Fig. 14. *Index* performance under studied topologies with 100 Mbps links.

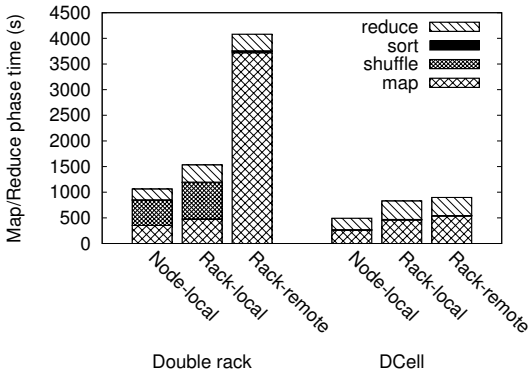


Fig. 15. Impact of data-locality on *TeraSort* performance.

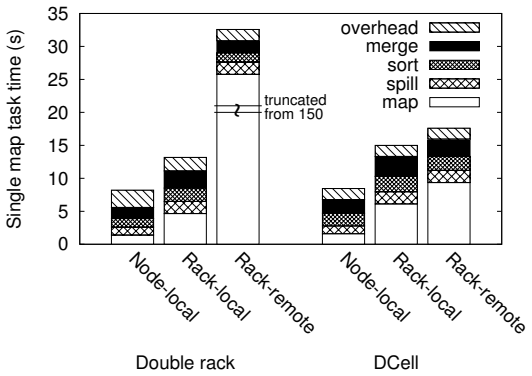


Fig. 16. Impact of data-locality on *TeraSort* map task sub-phases.

links produced similar results, and are not shown.

4) Effect on Index: Figure 13 and 14 show the effect of network topology on the performance of *Index* with 1 Gbps and 100 Mbps links, respectively. The simulations model 72-node topologies, 1 GB input data per node, and a 64 MB block size. Again, the needed input data for all map tasks is available locally.

These experiments show that the effect of network bottlenecks becomes much more pronounced as more intermediate data is generated by the map tasks, since all this data needs to be shipped across the network during the shuffle phase. Also, as before, switching to a 100 Mbps network exacerbates the problem and causes a larger spread in performance across topologies even in the case where maps output is only 10% of their input data.

Designers of MapReduce clusters should take these results into account and evaluate the most cost-effective ways of achieving acceptable network performance with all-to-all communications. As shown in this section, the characteristics of the applications that will be run on the cluster play an important role in predicting the demand on the networking infrastructure.

D. Impact of Data Locality

In this set of experiments, we evaluate how data locality affects application performance. For this purpose, we compare three different job scheduling decisions, which result in different data locality for the jobs. These localities are as follows. *Node-local* where all the needed data is available on the node and no remote retrieval is required. This occurs if sufficient data replication has been employed and the compute cluster overlaps with the data cluster. *Rack-local* where all the needed data is found within the rack but not on the node. We study this case since racks have good inter-node bandwidth, so scheduling tasks to access data within the rack is considered preferable to outside the rack. *Rack-remote* where all data has to be retrieved over the network from a remote rack. This can occur when a cluster is designed with separate compute and data sub-clusters, or if local map slots are not available on nodes containing the data when multiple jobs are run on a single cluster. For these experiments, we use the *Double rack*, *Tree* and *DCell* topologies.

1) Effect on TeraSort: Figure 15 shows overall execution time for *TeraSort*, where as Figure 16 shows the break-up of map phases (*Rack-remote* bar of *Double rack* is truncated). The most time is consumed by the map function as it involves remote data retrieval. We observe that data locality affects *Double rack* significantly, with execution time increasing by 284% for *Rack-remote* compared to *Node-local*. In contrast, *DCell* is able to provide better network bandwidth, and thus the results under this topology *Rack-remote* are similar to that of *Rack-local* and *Node-local*.

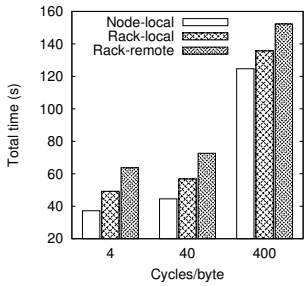


Fig. 17. Impact of data-locality on *Search* performance using *DCell*.

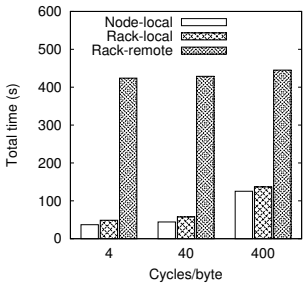


Fig. 18. Impact of data-locality on *Search* performance using *Double rack*.

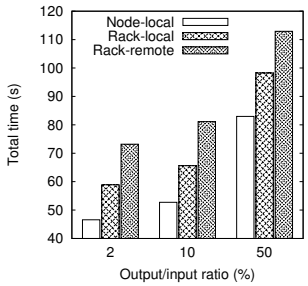


Fig. 19. Impact of data-locality on *Index* performance using *DCell*.

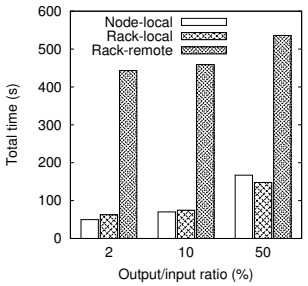


Fig. 20. Impact of data-locality on *Index* performance using *Double rack*.

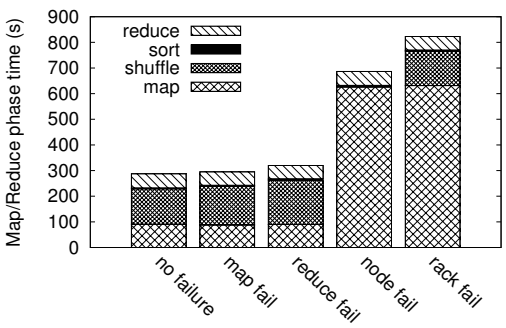


Fig. 21. *TeraSort* performance under failure scenarios.

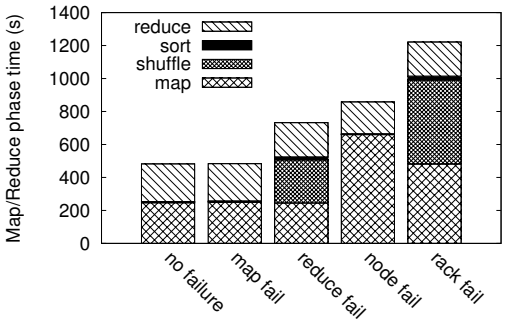


Fig. 22. *TeraSort* performance under failure scenarios using a 20-node cluster.

2) Effect on Search: Figure 17, and Figure 18, show the impact of different data-locality conditions on the performance of *Search* for the *DCell* and *Double rack* topologies, respectively. *Search* with more complex match criteria (and longer map times), and configurations with data farther from the compute node generally take longer, as expected. An interesting situation occurs in the *Double rack* case when the data is *Rack-remote*. Here the network latency during the map phase dominates execution time, and the execution time does not change with map phase CPU cost.

3) Effect on Index: Figure 19 and Figure 20, the impact of different data-locality conditions on performance of *Index* for the *DCell* and *Double rack* topologies, respectively. *Index* generates significant intermediate data (unlike *Search*), thus the network is shared for map data transfers and intermediate data shuffle. Again, the trends are as expected, with the applications that generate more intermediate data taking longer to complete, and faring worse in topologies where point-to-point bandwidth is lower.

E. Impact of Failures

In this set of experiments, we study how failures affect the performance of Hadoop applications. The failure scenarios that we consider are: (i) a map task fails; (ii) a reduce task fails; (iii) a node fails; and (iv) the inter-rack link fails (equivalent to a rack failure since it causes a network partition). Unless otherwise specified each experiment models a 72-node *Double rack* topology setup, and scheduling is such that node-local data locality is achieved.

1) Failure detection and recovery: The failure model in *MRPerf* mimics Hadoop's as follows. Task failures, i.e., map and reduce operation failures, are detected almost instantaneously by the local task tracker, and a failed task is re-started immediately upon detection. Such a failure results in loss of all the work done by the failed task. In contrast to task failures, a node or rack-level failure cannot be detected immediately. Instead, if the job tracker does not receive any messages from a node or rack for a pre-specified timeout period (default is 10 minutes in Hadoop), it infers that the non-responding unit has failed. Map task intermediate data stored on a failed node is considered lost. However, not all the map tasks need to be re-run for recovery, as some of it may already have been copied by reduce tasks running on different nodes. Thus, instead of trying to launch recovery immediately upon failure detection, we wait for reduce tasks corresponding to maps tasks on the failed node to report errors in reading necessary intermediate data, and only then re-start the failed map tasks. Although simple, such an on-demand recovery approach can result in delays in the recovery process. Finally, a rack-level failure is treated as multiple node failures in *MRPerf*, wherein all nodes in a failed rack are considered to have failed and their behavior is modeled as described above.

2) Effect on TeraSort: Figure 21 shows the overall execution time of *TeraSort* while failures occur during the execution. We observe that the Hadoop is able to tolerate a map task failure with negligible (<3% compared to no failures) effect. This is because a single map task represents a small fraction

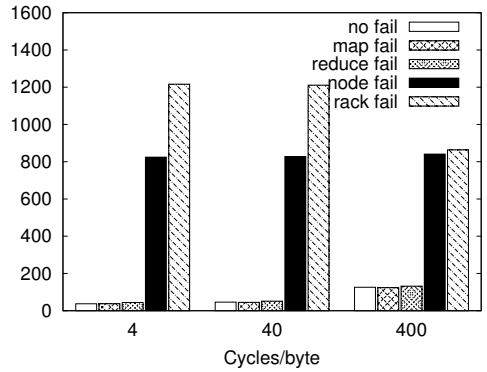


Fig. 23. *Search* performance under failure scenarios.

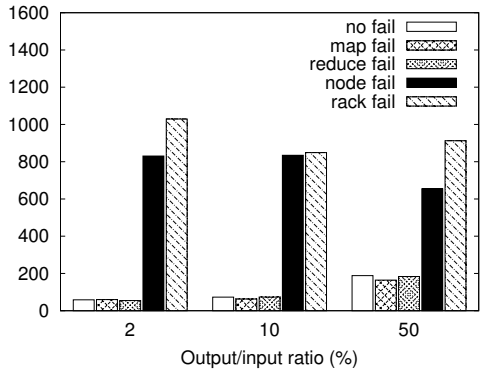


Fig. 24. *Index* performance under failure scenarios.

of the overall work, and the scheduler is able to re-run the failed map task without affecting any other tasks. This task isolation is a key benefit of MapReduce model. A reduce failure represents a larger fraction of work being lost, but with 72-node, some reduce tasks run slower than others, and the re-tried reduce task is able to finish faster because it competes for network bandwidth with only the slow reduce tasks. Thus, even a reduce failure is handled without significant performance penalty (11% slowdown). A node failure, and rack failure have much larger impacts on performance (139% and 186% respectively), partially because of the failure detection time-out (see Section IV-E1) and partially because of the larger loss of computation and intermediate data.

Figure 22 shows the results for the same failures for roughly the same amount of data sorted on a smaller cluster (20 nodes with 4 GB/node). Here we see that a reduce task failure results in 34% performance degradation. This is because there is smaller variability in the reduce times on this cluster and a reduce task loss means that $1/20$ of the shuffle and reduce steps have to be re-run. The perforamance degradation on a node failure is 44%, mainly due to the failure detection time-out and the lost data. The worst performance, i.e. 60% degradation, occurs when the intra-rack link fails. This is because when the two racks are separated, the entire job has to be re-run on one rack that contains the job tracker.

3) Effect on Search: Figure 23 shows impact of failures on *Search*. The effect of map and reduce task failures are small as is the case with *TeraSort*. However, the 10 minute failure detection time-out dominates the run-time of node and rack failure cases for this application due to the shorter total run-time. An interesting trend is that the longer running versions (i.e with larger cycles/byte) of *Search* actually finish faster in the case of rack failure. This is because the recovery time is longer if the number of completed map tasks on the failed rack is larger (see re-tries of map tasks in the case of node/rack failures in Section IV-E1).

4) Effect on Index: Figure 24 shows impact of failures on *Index*. Again, the cases of map and reduce task failures are similar to the previous applications. Also, for rack failure, the recovery for the 10% *Index* takes less time than the application with the 2% case due to the effect of failed map task recovery, similar to the rack failure recovery for *Search* experiments. However, the trend does not continue for the 50% case because higher shuffle requirements cancel out the faster recovery due to fewer failed map tasks.

These results are as expected given the base MapReduce design, and show the ability of *MRPerf* to capture Hadoop behavior under various failures. An important caveat to these results is the fact that *MRPerf* does not capture an important feature of Hadoop – speculative execution. Hadoop starts backup map and reduce tasks when it finds that execution slots are free and there are tasks which have taken longer than expected. We plan to add this feature to *MRPerf* in the future. The scheduling policies of speculative execution are a subject of active development in the Hadoop community, and we hope that including this capability in *MRPerf* will provide a way of systematically comparing different policies.

F. Summary of Results

MRPerf enables quantifying the affect of various design decisions on Hadoop applications. We have shown that advanced topologies such as *DCell* can help improve overall system performance. This stresses that cluster designers should consider such topologies while choosing networks for MapReduce clusters. Moreover, we have quantified the drastic effects that data-locality has on application performance. This stresses the need for prioritizing data locality in job scheduling decisions. We have also shown that MapReduce can tolerate failures in the map tasks and node failures with negligible or small impact, respectively, however, inter-rack link failures can reduce the performance significantly. Consequently, building redundancy into inter-rack connectivity may be necessary for mitigating the affects of failures.

We found it instructive to observe the inter-play of resource bottlenecks and scheduling decisions in determining the performance of Hadoop applications. We stress that studying this design space using actual clusters is next to impossible given equipment costs, extensive configurations and setup times, man-power needed, in-efficiency of the approach in terms of resources used and results obtained, and most importantly, the need to re-do the entire testing process for different clusters, applications, configurations. Thus, simulation is a powerful

and efficient approach in this context. This has led us to believe that *MRPerf* is an important tool for predicting the performance of applications on Hadoop platform.

V. Related work

Given that MapReduce is a relatively new programming model, it has not been previously studied through detailed simulations. However, a closely related large-scale distributed computing paradigm is Grid computing [15]. Grid computing is well-established 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 [16], Microgrid [17], Simgrid [18], and GridSim [19]. The interest in using simulation to model large-scale distributed systems can be gauged from the fact that the SourceForge project for GridSim shows over 5000 downloads between September 2007 and March 2009. Unlike these simulators for Grid, *MRPerf* is focused on modeling the specifics of MapReduce frameworks and 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 [20] and instrumenting Hadoop using X-Trace [21]. 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 projects, so that analysis tools developed for them can also be used to study results from *MRPerf*, and vice versa.

VI. Conclusion

We have discussed the design, evaluation, and application of *MRPerf*, a realistic phase-level simulator for the widespread MapReduce framework, toward designing, provisioning, and fine-tuning Hadoop setups. *MRPerf* provides means for analyzing application performance on a given Hadoop setup, and serves as a tool for evaluating design decisions for fine-tuning and creating Hadoop clusters. We have verified the simulator using a medium-scale cluster, and have shown that it effectively models MapReduce setups. Moreover, we applied *MRPerf* to study the impact of data locality, network topology and node failures on application performance, and have shown that network topology choices and scheduling decisions can have a large impact on performance. Thus, *MRPerf* can help in designing new high performance MapReduce setups, and in optimizing existing ones. Exploring Hadoop's design space using actual clusters is impractical given the in-efficiency of the approach in terms of resources used and results obtained, and the need to re-do the entire testing process for different clusters, applications, configurations. Thus, simulation is a powerful and efficient approach in this context. In summary, *MRPerf* provides a powerful system planning and design tool for researchers and IT professionals in realizing emerging MapReduce setups.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. of the ACM*, 51(1):107–113, 2008.

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

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

[4] Amazon. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[5] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proc. IEEE HPCA*, 2007.

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

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

[8] The Terasort benchmark, Jun 2009. http://sortbenchmark.org/.

[9] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: a scalable and fault-tolerant network structure for data centers. *SIGCOMM Comput. Commun. Rev.*, 38(4):75–86, 2008.

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

[11] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta. Using realistic simulation for performance analysis of mapreduce setups. In *Proc. LSAP*, 2009.

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

[13] Hadoop's implementation of the Terasort bench-mark, Mar 2009. http://hadoop.apache.org/core/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html.

[14] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *Proc. USENIX FAST*, 2008.

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

[16] Kento Aida, Atsuko Takefusa, Hidemoto Nakada, Satoshi Matsuoka, Satoshi Sekiguchi, and Umpei Nagashima. Performance Evaluation Model for Scheduling in Global Computing Systems. *Int. J. High Perform. Comput. Appl.*, 14(3):268–279, 2000.

[17] 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.

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

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

[20] Jerome Boulon, Andy Konwinski, Runping Qi, Ariel Rabkin, Eric Yang, and Mac Yang. Chukwa, a large-scale monitoring system. In *Proc. CCA*, 2008.

[21] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: A Pervasive Network Tracing Framework. In *Proc. USENIX NSDI*, 2007.