MRemu: An Emulation-based Framework for Datacenter Network Experimentation using Realistic MapReduce Traffic

Marcelo Veiga Neves, Cesar A. F. De Rose
Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre, Brazil
marcelo.neves@pucrs.br, cesar.derose@pucrs.br

Kostas Katrinis
IBM Research – Ireland
Dublin, Ireland
katrinisk@ie.ibm.com

***Abstract*—As data volumes and the need for timely analysis grow, Big Data analytics frameworks have to scale out to hundred or even thousands of commodity servers. While such a scale-out is crucial to sustain desired computational throughput/latency and storage capacity, it comes at the cost of increased network traffic volumes and multiplicity of traffic patterns. Despite the sheer reality of the dependency between datacenter network (DCN) and time-to-insight through big data analysis, our experience as active networking researchers conveys that a large fraction of DCN research experimentation is conducted on network traces and/or synthetic flow traces. And while the respective results are often valuable as standalone contributions, in practice it turns out extremely difficult to quantitatively assess how the reported network optimization results translate to performance or fault-tolerance improvement for actual analytics runtimes, e.g., due to the ability of these runtimes to overlap communication with computation. This paper presents MRemu, an emulation-based framework for conducting reproducible datacenter network research using accurate MapReduce workloads and at system scales that are relevant to the size of target deployments, albeit without requiring access to a hardware infrastructure of such scale. We choose the MapReduce (MR) framework as a design point, for it is a common representative of the most widely deployed frameworks for analysis of large volumes of - structured and unstructured - data and is reported to be highly sensitive to network performance. With MRemu, it is possible to quantify the impact of various network design parameters and software-defined control techniques to key performance indicators of a given MR application. We show through targeted experimental validation that MRemu exhibits high fidelity, when compared to the performance of MR applications on a real scale-out cluster of 16 high-end servers. Also, as a proof of impact of our experimental framework, we showcase how we used MRemu to quantify the impact of network capacity among MR nodes to a selection of network-bound MR applications.**

I. Introduction

The rise of Internet of Things sensors, social networking and mobile devices has led to an explosion of data available for analysis towards knowledge gaining and final insights. In turn, this has led into the development of dedicated platforms for large-scale data analysis. The MapReduce (MR) framework, as implemented in Hadoop, is one of the most popular frameworks for Big Data analysis. To handle the ever-increasing data size, Hadoop is a scalable framework that allows a dedicated and seemingly unbound number of servers to participate in the analytics process. The response time of an analytics request is an important factor for time to value/insights. While the computing and disk I/O requirements can be scaled with the number of servers, scaling the system leads to increased network traffic. Evidently, the communication-heavy phases of MR contributes significantly to the overall response time [1].

Despite Big Data analytics being a very common workload in datacenters, most research on datacenter networks does not really take it into consideration. Instead, researchers normally use synthetic traffic patterns (e.g., random traffic following probabilistic distributions) to evaluate network performance [2], [3], [4], [5]. While useful to rapidly evaluate new algorithms and techniques, this approach often fails to capture the real strain on datacenter networks supporting MR-like runtimes. Thus, it is difficult to determine how these studies would perform in the presence of MR workloads and, most importantly, how they impact MR response time. In fact, recent research has shown that MR applications are sensitive to network performance [6], [1] and that it is possible, by leveraging the emergent technology of software-defined networks (SDN), to develop network control software to adapt the network to the application's needs in order to accelerate the execution of this kind of application [6], [7].

Some studies in the literature use MapReduce-like traffic patterns to evaluate their research [8], [9], [10], [5]. They normally model the MR shuffle pattern as map tasks (typically one per node) sending data to one or more reduce tasks. However, MapReduce frameworks, such as Hadoop, implement some mechanisms to improve network performance that are not taken into account by these works (e.g., transfer scheduling decisions, number of parallel transfers, etc.). Moreover, normally there are a large number of map tasks per node and Hadoop nodes have to serve data to multiple reduce tasks concurrently. As a result, a great deal of research is being conducted using synthetic traffic patterns and may not perform well when applied to real MapReduce applications, particularly those that claim to improve MapReduce performance.

Aside from the use of unrealistic network traffic patterns, most research in this area focuses on maximizing aggregate network utilization [2], [3], [4], [5], which may not be the best metric when considering MR applications. High network utilization does not necessarily ensure shorter job completion times, which have a direct impact on applications' response times. MR has some implicit barriers that depend directly on the performance of individual transfers. For example, a reduce task does not start its reduction phase until all input

data becomes available. Thus, even a single transfer being forwarded through a congested path during the shuffle phase may delay the overall job completion time. The problem is further aggravated if communication patterns are heavily skewed, which often happens in many MR workloads [11]. Research in this area often relies on analytical and simulation models to evaluate the network performance, which may not capture all of the characteristics of real networks and applications. The use of real hardware, on the other hand, is often not a valid option, since many researchers do not have access to datacenters robust enough to run data-intensive applications. Moreover, even when datacenter resources are available, it is normally not practical to reconfigure them in order to evaluate different network topologies and characteristics (e.g., bandwidth and latency). In this context, an alternative is to use emulation-based testbeds. In fact, network emulation has been successfully used to reproduce network research experiments with a high level of fidelity [12]. Although existing network emulation systems allow for the use of arbitrary network topologies, they typically run on a single physical node and use some kind of virtualization (e.g., Linux containers) to emulate the datacenter nodes. Therefore, they lack resources to run real Hadoop jobs, which are known to be CPU and IO-intensive.

To overcome the above gap, we propose to combine network emulation with trace-driven MapReduce emulation. For this, we have implemented a framework called MRemu that reproduces executions of MapReduce jobs by mimicking Hadoop MapReduce internals (e.g., scheduling decisions) and generating traffic in the same way a real Hadoop job would. The proposed framework enables networking research experiments targeting datacenter networks using SDN and MapReduce-like workloads, however by slashing the requirements of continuous access to a large infrastructure. For example, it is possible to compare the performance (e.g., job completion time) of a given MapReduce job on different network topologies and network control software. The MapReduce emulation also works as a stand-alone tool that can be used to run network experiments in clusters with limited resources (i.e., not robust enough to run real Hadoop jobs [9]). MRemu sports following features and functionality:

- Ability to create arbitrary network topologies, including complex multi-path topologies (e.g., fat-tree), and network parameters (e.g., bandwidth, latency, delay, and packet-loss ratio);
- Accurate reproduction of MapReduce workloads, including jobs with skewed transfer patterns, without requiring a real datacenter infrastructure;
- Support for evaluating real software-defined network (SDN) control code running on production SDN controllers, through interfacing MRemu with SDN controllers via the OpenFlow protocol and thus slashing "from-lab-to-market" integration times. We have successfully tested MRemu working in tandem with production SDN controllers, notably OpenDaylight [13] and POX/NOX [14].

This paper is structured as follows. Section II outlines background information related to MapReduce and associated data movement patterns. Section III presents the proposed testbed, including network emulation framework and MapReduce workload generation. Sections IV and V present evaluation results manifesting the accuracy and impact of our approach and framework embodiment. We then review and put our work in context of related work in the Section VI. Conclusions and future work are presented in Section VII.

II. Background

This section first provides an overview of the MapReduce model and the Hadoop MapReduce implementation. Although there are currently several implementations that are functionally equivalent to MapReduce (e.g., Hadoop [15], Dryad [16], Twister [17], Spark [18]), this work will focus on Hadoop because it is one of the most popular open-source implementations for Big Data analysis implementations.

A. MapReduce Model

The MapReduce programming model was first introduced in the LISP programming language and later popularized by Google [19]. It is based on the *map* and *reduce* primitives, both to be provided by the programmer. The *map* function takes a single instance of data as input and produces a set of intermediate key-value pairs. The intermediate data sets are automatically grouped based on their keys. Then, the *reduce* function takes as input a single key and a list of all values generated by the *map* function for that key. Finally, this list of values is merged or combined to produce a set of typically smaller output data, also represented as key-value pairs.

MR implementations are typically coupled with a distributed file system (DFS), such as GFS [19] or HDFS [20]. The DFS is responsible for the data distribution in a MR cluster, which consists of initially dividing the input data into blocks and storing multiple replicas of each block on the cluster nodes' local disks. The location of the data is taken into account when scheduling MR tasks. For example, MR implementations attempt to schedule a map task on a node that contains a replica of the input data. This is due to the fact that, for large data sets, it is often more efficient to bring the computation to the data, instead of transferring data through the network. After the reduction phase, the output data is persisted to the DFS.

B. Hadoop

The Hadoop framework can be roughly divided in two main components: the Hadoop MapReduce, an open-source realization of the MapReduce model and the Hadoop Distributed File System (HDFS), a distributed file system that provides resilient, high-throughput access to application data [20]. The execution environment includes a job scheduling system that coordinates the execution of multiple MapReduce programs, which are submitted as batch jobs.

A MR job consists of multiple map and reduce tasks that are scheduled to run in the Hadoop cluster's nodes. Multiple jobs can run simultaneously in the same cluster. There are two types of nodes that control the job execution process: a JobTracker and a number of TaskTrackers [21]. A client submits a MR job to the JobTracker. Then, the JobTracker, which is responsible for coordinating the execution of all the jobs in the system, schedules tasks to run on TaskTrackers, which have a fixed number of slots to run the map and reduce

tasks. TaskTrackers run tasks and report the execution progress back to the JobTracker, which keeps a record of the overall progress of each job. The client tracks the job progress by polling the JobTracker.

Each data block is independently replicated (typically 3 replicas per block) and stored at multiple yet distinct DataNodes. Replicas placement follows a well-defined rack-aware algorithm that uses the information of where each DataNode is located in the network topology to decide where data replicas should be placed in the cluster. Effectively, for every block of data, the default placement strategy is to place two replicas on two different nodes in the same rack and the third one on a node on a different rack. Replication is used not only for providing fault tolerance, but also to increase the opportunity for scheduling tasks to run where the data resides, by spreading replicas out on the cluster. For example, if a node that stores a given data block is already running too many tasks, it is possible to ingest this same data block on another node that holds one of the block's replicas. Otherwise, if there is no opportunity to schedule the task locally, the data block must be read from a remote node, which is known to degrade job performance due to increased data movement.

Assuming that the input data is already loaded into the distributed file system (i.e. excluding data movement due to loading/exporting data to/from HDFS), intensive data movement in Hadoop is mainly attributed to shuffling intermediate mapper output to reducers. For instance, a recent analysis of MR traces from Facebook revealed that 33% of the execution time of a large number of jobs is spent at the MR phase that shuffles data between the various data-crunching nodes [1]. This same study also reports that for 26% of Facebooks MR jobs with reduce tasks, the shuffle phase accounts for more than 50% of the job completion time, and in 16% of jobs, it accounts for more than 70% of the running time.

Hadoop Data Shuffling. In the shuffle phase of a MR job, each reduce task collects the intermediate results from all completed map tasks. Reduce tasks are normally scheduled after a fraction of map tasks have been completed (by default 5%). Once running, a reduce task does not wait for all map tasks to be completed to start fetching map results. Instead, it starts scheduling copier threads to copy map output data as soon as each map task commits and the data becomes available. This technique (often referred to as early shuffle [11]) causes the overlap between the execution of map tasks and the shuffle phase, which typically reduces the job completion time. However, the reduction itself starts only after all map tasks have finished and all intermediate data becomes available, which works as an implicit synchronization barrier that is affected by network performance.

Despite the well-defined communication structure of the shuffle phase, the amount of data generated by each map task depends on the variance of intermediate keys' frequencies and their distribution among different hosts. This can cause a condition called partitioning skew, where some reduce tasks receive more data than others, resulting in unbalanced shuffle transfers [11]. Since a reduce task does not start its processing until all input data becomes available, even a single long transfer in the shuffle phase can delay the overall job completion time. Recent work has reported [22] that the amount of data exchanged during the shuffle phase of MR jobs from Facebook and Yahoo! can vary from tens of megabytes to hundreds of gigabytes, with a few jobs exchanging up to 10 TB. It is thus of paramount importance to evaluate the impact of network capacity/adaptation to MR jobs with significant data skew.

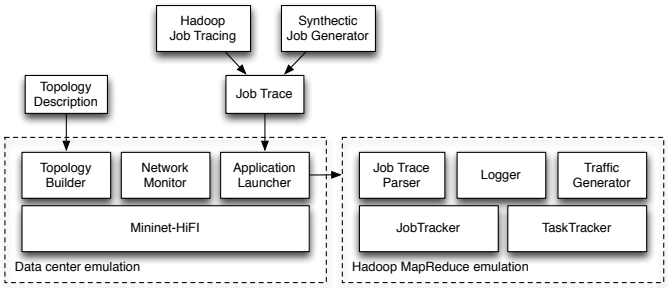

Fig. 1. Block diagram showing MRemu architecture.

In general, the number of map tasks within a MR job is driven by the number of data blocks in the input files. For example, considering a data block size of 128 MB, a MR job with an input data of 10 TB will have 82K map tasks. Therefore, there are potentially many more map tasks than task slots in a given cluster, which forces tasks to run in waves [23]. The number of reducers, on the other hand, is typically chosen small enough so that they all can launch immediately enabling the early shuffle technique [11], as previously described. Thus, reduce tasks normally have to copy output data from tasks from different nodes and wave generations. These transfers can be performed in parallel, but Hadoop limits the number of parallel transfers per reduce tasks (by default 5) to avoid the so called TCP Incast [24] effect. The algorithm used by Hadoop to schedule these shuffle transfers is detailed in Section III-C.

III. The MREmu Framework

This section describes the design and architecture of the proposed emulation-based experimentation framework to enable networking experiments with MapReduce applications. As introduced earlier, the main goals of this work are (1) to enable the evaluation of network design parameters to MapReduce-like application performance (e.g., job completion time), (2) to enable the evaluation of novel software-defined datacenter network control algorithms and policies to MapReduce-like applications and (3) to provide for a means of identifying network bottlenecks and optimization opportunities caused by new MapReduce-like applications (e.g. workflows comprising inter-dependent MR jobs). For this, we decided to combine network emulation with trace-driven MapReduce emulation, as conveyed in the block diagram of Figure 1 showing the functional blocks of the MRemu framework. The rest of this section will explain each of the proposed system's components in detail.

A. MapReduce Job Tracing

The emulation-based experiments are driven by job traces that can be either extracted from past job executions or synthetically generated by statistical distributions. Hadoop has a built-in tool, called Rumen [25], that generates job trace files for past executions. However, the trace data produced is insufficient for network-related analysis, since it lacks information about individual network transfers. Thus, we developed a

new tool for extracting meaningful information from Hadoop logs (including network-related information) and generating comprehensive MR execution traces. This tool is also able to create job traces with different numbers of nodes and tasks by using linear extrapolation, which is particularly useful for scaling experiments for setups larger than those where the job traces were collected from.

For every job executed, Hadoop creates a JobHistory log file containing task-related information, such as the task's start/finish times, amount of data transferred/processed by each task, etc. Additionally, each Hadoop daemon (e.g., JobTracker, TaskTracker and DataNode) registers important events to its local log files, including individual transfers' end/duration/size. Our tool combines information from JobHistory and Hadoop daemons' logs to produce a single job trace file containing not only the usual task-related information, but also information about network transfers, distributed file systems operations and Hadoop configuration parameters. Since Hadoop is often configured for rack-awareness, our tool is also able to infer the network topology based on host locations and rack IDs that arises in the JobTracker log. The generated job trace and topology files have an easily-parsed format (using a JSON format [26]), similar to Rumen.

B. Data Center Emulation

This work relies on Mininet-HiFi [12] for network emulation. Mininet-HiFI, also called Mininet 2.0, is a container-based network emulation tool designed to allow reproducible network experiments. It extends the original Mininet [27] with features for performance isolation, resource provisioning, and performance fidelity monitoring. Performance isolation and resource provisioning are provided using Linux containers features (e.g, cgroups and namespaces) and network traffic control (tc). Fidelity is achieved by using probes to verify that the amount of allocated resources were sufficient to perform the experiment.

We have developed a tool for automatically setting up a datacenter network experiment scenario, launching the experiment code and monitoring performance information. It consists of three main components: TopologyBuilder, ApplicationLauncher and NetworkMonitor. The TopologyBuilder component ingests a datacenter network description file, which can be either the topology file extracted from Hadoop traces (as described in Section III-A) or a manually created file supporting the use of any arbitrary network topology, and uses Mininet's Python APIs to create a complete network topology. ApplicationLauncher is an extensive component that launches the application code in each emulated node and waits for its completion. Currently, it supports two applications: the MapReduce emulator (as described in Section III-C) and a synthetic traffic generator using iperf (as described in Section V-C). Finally, NetworkMonitor allows for probes to be installed in every node or network element to collect monitoring information. Examples of information that can be collected are network bandwidth usage, queue length, number of active flows, CPU usage, etc.

The emulated network can be composed of both regular Ethernet switches and OpenFlow-enabled switches. For OpenFlow-enabled switches, it is also possible to connect to external network controllers that can be running locally or in a remote host. This last option is especially useful to test real SDN control code. The same code running in a real hardware infrastructure can be tested in the emulated one without modifications. Similarly, the code developed using this platform can be easily deployed in a real infrastructure. We have successfully tested this indispensable functionality against both the NOX/POX [14] and the OpenDaylight [13] network controllers.

Mininet-HiFi has been successfully used to reproduce the results of a number of published network research papers [28]. Although it can reproduce the results, sometimes it is not possible to reproduce the exact experiment itself, or at least not at the same scale, due to resource constraints [12]. For example, a network experiment that originally uses 10Gbps (or even 1Gbps) links cannot be emulated in real time on a single host running Mininet. The alternative in this case is to scale down the experiment by emulating 10 or 100Mbps links. If no explicit configuration is provided, our tool automatically computes the scale down ratio between the real hardware bandwidth (i.e., the one where execution traces were extracted from) and the local Mininet emulation capacity. Otherwise, a configuration file is used to allow for specifying any arbitrary bandwidth.

Although the scale down approach has been proven to work [12], [28], the maximum size of network topology that can be emulated is clearly limited by the server's capacity. To overcome this limitation, a Mininet Cluster Edition is currently under development [29]. It will support the emulation of massive networks of thousands of nodes. By running Mininet over a distributed system, it will utilize the resources of each machine and scale to support virtually any size of network topology. Integrating it against our framework - as soon as Mininet Cluster Edition becomes available through it upstream distribution - constitutes one of our high-priority tasks for extending MRemu.

C. Hadoop MapReduce Emulation

We implemented a tool that reproduces executions of MapReduce jobs by mimicking Hadoop MapReduce internals (e.g., scheduling decisions) and generating traffic in the same way a real Hadoop job does. Although it internally simulates part of MapReduce functionality, it constitutes a MapReduce emulation tool from the system networking point of view: to the network system, our evaluation tool has exactly the same effect as a real MapReduce application producing real network traffic and logging events to local files. This also allows, for example, plugging systems that extract information from Hadoop logs to predict network transfers [6], [7], [30]. Since we are interested mainly in the MapReduce shuffle phase, simulating the details of Hadoop daemons and tasks (e.g., task processing internals, disk I/O operations, control messages, etc.) is not a goal, unlike MapReduce simulators such as MRPerf [31] and HSim [32]. Instead, we use information extracted from job traces (e.g., task durations, wait times, etc.) to represent the latencies during different, non-shuffling phases of the MapReduce processing.

The MapReduce emulator loads a job trace file (extracted from past Hadoop executions) and a network topology description and extracts all the parameters necessary for the simulation, which occurs as follows. First, the emulator starts

a JobTracker on the master node and a TaskTracker, one each on each of the slave nodes. The JobTracker then receives a job submission request and signals all TaskTrackers to start executing map tasks on their task slots. The system then simulates task scheduler decisions, task execution durations, and execution latencies, based on time parameters extracted from the job trace file. The first reduce task is scheduled when 5% (configurable) of the map tasks have finished and starts fetching map output data as soon as the latter becomes available. For this, each reduce task starts a number of ReduceCopier threads (the exact number is defined by the Hadoop parameter $max.parallel.transfers$) that copies map output data from a completed map tasks. This process truthfully emulates the shuffle phase and triggers real data transfers over the underlying network (we have tested data transfers using both the iperf tool and our in-house developed tools and have obtained similar results).

Our emulator implements two operation modes: REPLAY, which reproduces the exact scheduling decisions from MR application execution log traces, and HADOOP, which computes new scheduling decisions based on the same algorithms that Hadoop uses to schedule jobs and transfers used by Hadoop. As the name conveys, the first mode allows us to reproduce the exact order and timing of individual transfers, as these occurred during execution of the MR application on a real scale-out machine. The second mode may not reproduce the same order and execution timing, but allows us to employ trace-driven evaluation by varying the network parameters and network control logic, without having to replicate the respective network configuration of the datacenter that the traces were collected from. Each reduce task in Hadoop schedules shuffle transfers as described by the pseudo-code in Algorithm 1 in a simplified and yet realistic way (failure and error handling are omitted for brevity).

Shuffle scheduling in Hadoop abides by the producer-consumer paradigm. Copier threads will consume map output tasks from a list of scheduled copies ($scheduledCopies$) as soon as they become available. The number of copier threads will determine the maximum number of parallel transfers/copies per reduce task. The producer part is implemented as described in Algorithm 1 and runs iteratively. At each iteration, each reduce task first obtains a list of events of completed map tasks received by the reducer-local TaskTracker since the last iteration. Then, it groups the completed map tasks' location descriptors per host and randomizes the resulting host list. This prevents all reduce tasks from swamping the same TaskTracker. Finally, it appends map tasks' location descriptors to the scheduled copies list that will be consumed by copier threads. Hadoop keeps track of hosts in the scheduled list ($uniqueHosts$) to make sure that only one map output will be copied per host at the same time. The size of the scheduled copies list is also limited ($maxInFlight$), so that the maximum number of scheduled transfers is defined as four times the number of copier threads.

It is worth mentioning that the Hadoop emulation system can also be used as a standalone tool. Although we have developed MRemu to run in container-based emulation environments, it is also possible to use it to perform experiments on real hardware testbeds that are not robust enough (due to ,e.g., lack of memory, compute or I/O rate) to execute real large-scale data-intensive jobs [9].

Algorithm 1 MapReduce shuffle scheduling

```
1:  numMaps ← Number of map tasks
2:  numCopiers ← Number of parallel copier threads
3:  numInFlight ← 0
4:  maxInFlight ← numCopiers × 4
5:  uniqueHosts ← {}
6:  copiedMapOutputs ← 0
7:  for i in numCopiers do
8:     init new MapOutputCopier() thread
9:  end for
10: while copiedMapOutputs < numMaps do
11:    mapLocList ← getMapComplEvents(mapLocList)
12:    hostList ← getHostList(mapLocList)
13:    hostList ← randomize(hostList)
14:    for all host in hostList do
15:       if host in uniqueHosts then
16:          continue
17:       end if
18:       loc ← getFirstOutputByHost(mapLocList, host)
19:       schedule shuffle transfer for loc
20:       uniqueHosts.add(host)
21:       numInFlight ← numInFlight + 1
22:       numScheduled ← numScheduled + 1
23:    end for
24:    if numInFlight == 0 and numScheduled == 0
       then
25:       sleep for 5 seconds
26:    end if
27:    while numInFlight > 0 do
28:       result = getCopyResult()
29:       if result == null then
30:          break
31:       end if
32:       host ← getHost(result)
33:       uniqueHosts.remove(host)
34:       numInFlight ← numInFlight - 1
35:       copiedMapOutputs ← copiedMapOutputs + 1
36:    end while
37: end while
```

D. Limitations

As will be shown in next sections, the proposed system has been successfully used to accurately reproduce MapReduce traffic and allows for realistic network experiments. Still, as is the case with any engineering artifact, our embodiment builds on design choices that trade-off full emulation fidelity for implementation complexity and reasonable emulation turnaround times. The current version of MRemu exhibits following limitations:

- Concurrent job execution: currently we support emulation of one job at a time. This is useful for evaluating the performance of a given MR job on different network topologies and network control software, but it does not take into consideration the dependencies between multiple jobs. Multi-job support is one of the first features we plan to add in the future. Still, it is currently possible to simulate multi-job impact to datacenter network performance through background traffic generation.

- Failure handling: currently, we do not model Hadoop failure handling techniques. Thus, it is required that the job traces used as input have been extracted from successfully executed jobs. By default, the system is able to identify traces with failures or missing information and abort its execution. Alternatively, it is still possible to fill missing information using the same technique as used for scaling up experiments.
- Distributed emulation: The size of each experiment (in terms of number of nodes and tasks per node) is limited by the amount of resources available at the emulation host. With this setup, we have been able to emulate datacenters with hundreds of nodes and tens of tasks per node. However, as mentioned before, the next version of Mininet-HiFi are expected to support distributed execution, which will greatly improve the emulation capacity of our system.

While reporting on limitations is of significant importance to our conduct, it must be noted that all of the limitations outlined are not fundamental and we plan to prioritize on overcoming them in future MRemu implementation iterations.

IV. EVALUATION

This section describes the experiments conducted to evaluate the fidelity of the MRemu implementation. Since Mininet-HiFi has already been validated [12] and is widely used to reproduce networking research experiments [28], we focus on the evaluation of our MapReduce emulation tool and its ability to accurately reproduce MapReduce workloads.

A. Experimental Setup

Our experiments are based on job traces extracted from executions in a real cluster. These traces are used both as input for our emulation system and baseline to evaluate the emulation results accuracy. The cluster setup we have used consists of 16 identical servers, each equipped with 12 x86_64 cores, 128 GB of RAM and a single HDD disk. The servers are interconnected by an Ethernet switch with 1Gbps links. In terms of software, all servers run Hadoop 1.1.2 installed on top of Red Hat Enterprise Linux 6.2 operating system. The emulation-based experiments were executed on a single server with 16 x86_64 cores at 2.27GHz and 16 GB of RAM. Our emulation tool runs within Mininet 2.1.0+ on top of Ubuntu Linux 12.04 LTS.

Since our Hadoop cluster setup has only one HDD disk per server (with measured serial read rate of 130MBytes/sec) and multiple cores accessing it in parallel, we decided to configure Hadoop to store its intermediate data in memory. Otherwise, Hadoop would operate in a host-bound range (i.e., disk I/O rate would be the bottleneck), thus resulting in the setup being indifferent to any improvement brought by network optimization. Having a balance between CPU, memory, I/O and network throughput is key in production-grade Hadoop clusters [33] and therefore following the above practice is justified in the absence of Hadoop servers with arrays of multiple disks.

B. Application Benchmarks

In order to evaluate our system, we selected popular benchmarks as well as real applications that are representative of significant uses of MapReduce (e.g., data transformation, web search indexing and machine learning). All selected applications are part of the HiBench Benchmark Suite [34], which includes Sort, WordCount, Nutch, PageRank, Bayes and K-means. As part of our previous work [7], we have established through tedious network profiling that following HiBench applications are not communication extensive: WordCount and K-means. As such, we exclude the latter from our MRemu evaluation and instead focus on following HiBench MR applications:

- Sort is an application example that is provided by the Hadoop distribution. It is widely used as a baseline for Hadoop performance evaluations and is representative of a large subset of real-world MapReduce applications (i.e., data transformation). We configure the Sort application to use an input data size of 32GB.
- The Nutch indexing application is part of Apache Nutch [35], a popular open source web crawler software project, and is representative of one of the most significant uses of MapReduce (i.e., large-scale search indexing systems). We configured Nutch to index 5M pages, amounting to a total input data size of $\approx$ 8GB.
- PageRank is also representative of large-scale search indexing applications. In particular, PageRank implements the page-rank algorithm [34] that calculates the rank of web pages according to the number of reference links. We used the PageRank application provided by the Pegasus Project [36]. It consists of a chain of Hadoop jobs that runs iteratively (we report sizes only for the most network-intensive job in the chain, namely Pagerank_Stage2). We configured PageRank to process 500K pages, which represents a total input data size of $\approx$ 1GB.
- The Bayes application is part of the Apache Mahout [37], an open-source machine learning library built on top of Hadoop. It implements the trainer part of Naive Bayesian, which is a popular classification algorithm for knowledge discovery and data mining [34]. Thus, it is representative of other important uses of MapReduce (i.e., large-scale machine learning). It runs four chained Hadoop jobs. We configured Bayesian Classification to process 100K pages using 3-gram terms and 100 classes.

C. Evaluation of the Job Completion Time Accuracy

In order to evaluate the accuracy of our Hadoop emulation system, we performed a comparison between job completion times in real hardware and in the emulation environment. We tested both REPLAY and HADOOP operation modes. The graph in Figure 2 shows normalized job completion times compared to those extracted from the original traces. As can be observed, job completion times for emulation in the REPLAY mode are very close to the ones observed in the original execution traces. When using the HADOOP mode (i.e., the one that computes new scheduling decisions, instead of directly

using times extracted from traces), the completion times are are slightly deviating for the Bayes application.

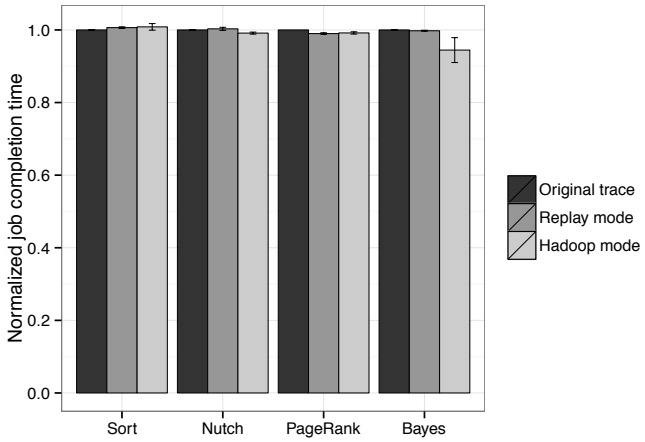

Fig. 2. Comparison between job completion times in real hardware and in the emulation environment for different MapReduce applications.

D. Evaluation of Individual Network Flow Completion Time Accuracy

After validating the system accuracy in terms of job completion time, we conducted experiments to evaluate individual flow durations. The graph in Figure 3 shows the Cumulative Distribution Function (CDF) for completion times of flows during the shuffle phase of the Sort application. We compared the results of the emulated execution with times extracted from the original execution trace. Although the mean time is very close, we can see in Figure 3 that the transfers' durations were slightly different. This behavior is expected since we are inferring flow durations from Hadoop logs, which may not represent the exact system behavior due to high-level latencies. Especially for small transfers, we detected that our traffic generation imposes an overhead, since it needs to start two new processes (client and server) at each new transfer (we plan to improve this in future). Nevertheless, the system still achieved completion times very close to the ones extracted from Hadoop traces and, as shown in Section IV-C, it leads to accurate job completion times.

E. Evaluation in the Presence of Partition Skew

We also conducted experiments to evaluate the ability of our system to accurately reproduce Hadoop job executions with skewed partitions. As explained in Section II-B, such a phenomenon is not uncommon in many MapReduce workloads and can be detrimental to job performance. This creates an obvious incentive for new research in this area, including optimizations via an appropriate dynamic network control. Therefore, one of the goals of this work is to verify whether our system could be used to conduct network research to overcome this problem.

To reproduce the partition skew problem in the Sort application, we modified the RandomWriter application in Hadoop to generate data sets with non-uniform key frequencies and data distribution across nodes. By using an approach similar to Hammoud et al. [11], we produced data sets with different

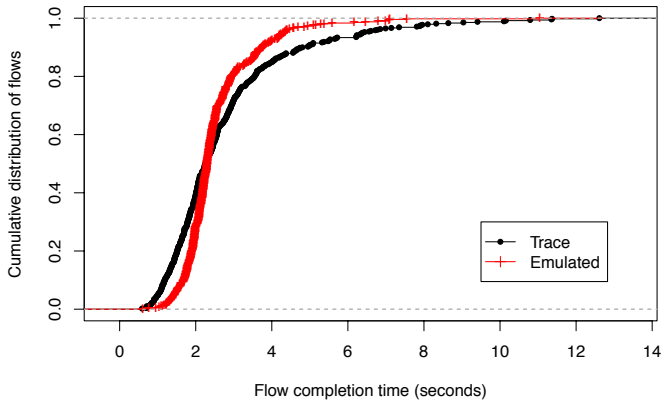

Fig. 3. Comparison of the cumulative distribution of flows durations in emulated execution and original traces.

key frequencies and data distribution variances. The graph in Figure 4 shows a comparison between the results of emulated jobs and results extracted from the original traces for different coefficients of variation in partition size distribution; from 0% (uniform distribution) to 100% (highly skewed partitions). As can be observed, job completion times for emulated jobs are very close to those observed in the original execution traces for all cases.

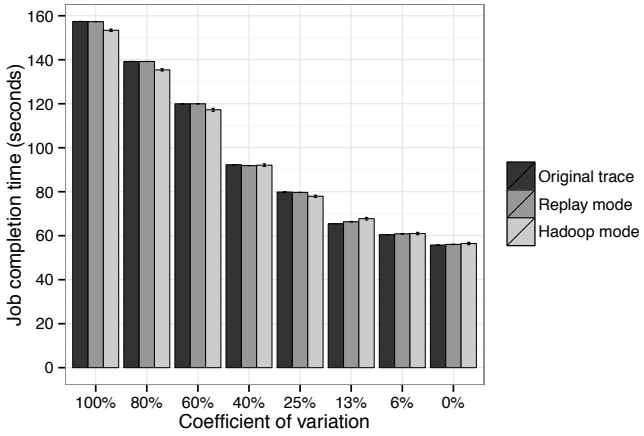

Fig. 4. Comparison between job completion times in real hardware and in the emulation environment for the Sort application with skewed partitions.

F. Experiments using Legacy Hardware

We also evaluated our Hadoop emulation tool as a standalone tool to be used to run network experiments in clusters with limited resources (i.e., not robust enough to run real Hadoop jobs [9]). For this, we ran our Hadoop emulation system on top of a 10+ year old computing cluster, composed of dual-core AMD Opteron(tm) processors at @ 2 GHz with 4 GB of RAM and a single HDD. The computing nodes were interconnected by a 1Gbps Ethernet switch. It is worth mentioning that such a cluster setup would never be able to

run network experiments using real Hadoop cluster installation to process large data sets, because the network performance would be limited by the host's capacity (e.g., CPU, memory and disk). Despite this limitation, we were able to reproduce the execution of the MR Sort application with a high level of accuracy (a difference of less than 2% when compared to the original trace's job completion times).

V. Case-Study: Using MRemu to evaluate the impact of key network characteristics to MapReduce applications

This section showcases the use of the MRemu framework to study the effect of various network design choices (link speed, network topology) and network characteristics (background traffic pattern) to tangible MapReduce applications. The value of this is straightforward, allowing a networking researcher, e.g., to obtain preliminary results for data-intensive workloads on top of an exploratory network topology she is exploring for future datacenters or have a network engineer test the impact of a new flow-path allocation algorithm implemented on a production SDN controller to the Hadoop deployment of its enterprise.

A. Link Bandwidth

We first conducted experiments aiming to verify the relationship between network bandwidth and performance in Hadoop jobs. For this, we executed the same MapReduce job, each time varying the available network bandwidth. We used a non-blocking network (i.e., a single large-switch network) to make sure that hosts are not limited by the network topology (e.g., oversubscribed network links). It is worth mentioning that such an experiment would not be entirely possible in our original hardware setup, for it comprises 1Gbps network links solely. Figure 5 shows the job completion time of the MR Sort application for different network bandwidths. We observe that bandwidth has a high impact to job performance. For example, when we halve link bandwidth from 1Gbps to 500Mbps, performance decreases by more than 1.5x; but, when link bandwidth is reduced to 100Mbps, the job is experiencing more than 5x slowdown. The impact is also observed when we provided more bandwidth to the application. For example, when we increased link bandwidth from 1Gbps to 2Gbps, the application is 26% faster. Job completion time continues to decrease as more bandwidth is provided (up to 10Gbps in this example). Assuming a perfectly balanced system (compute-network-I/O), we expect this trend to be preserved when executing the same job in a real datacenter environment.

B. Network Topology

In the previous sub-section, we evaluated the impact of network bandwidth, assuming a fully non-blocking network. Here we employ the opposite view: we showcase the use of MRemu in evaluating the impact of datacenter network topology to MR Sort. Specifically, we evaluate MapReduce job performance in three different network topologies: a dumbbell-like topology (2 switches), a single-path tree (depth 2, fanout 4) and a multipath tree organized as a 4-ary fat-tree. We use an ECMP-like technique to distribute flows to possible source-destination paths (relevant for the multipath topology only). Figure 6 shows emulated job completion times in each of the three network topologies tested. We also report results obtained in a fully non-blocking network as the baseline, since it represents the case where the application is not limited by the network topology. Following intuition, the results verify that job completion time is proportional to network topology bisection bandwidth, with the fat-tree topology outperforming the single path alternatives.

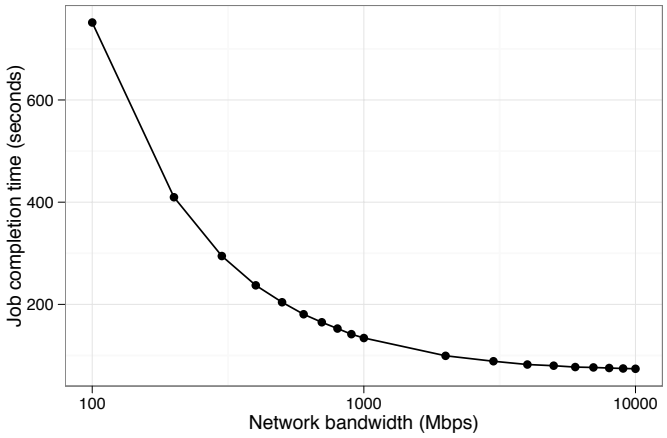

Fig. 5. Impact of (uniform) link bandwidth to MR Sorting (fully non-blocking network)

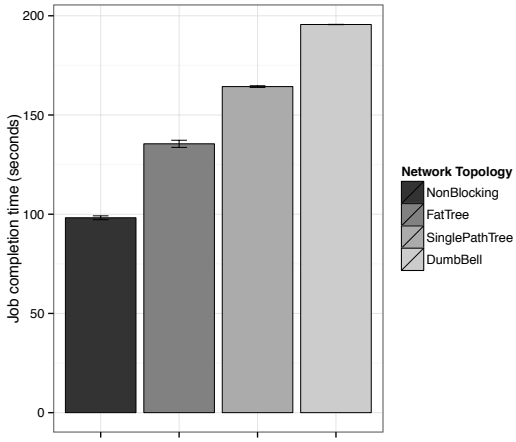

Fig. 6. Job completion time for different network topologies.

C. Background Traffic

In this section we evaluate MapReduce performance, as a function of the background traffic that the datacenter network exhibits. We performed all background traffic experiment in a 4-ary fat-tree network topology and an ECMP-like network control to distribute flows among the multiple paths. Since there are no commercial datacenter traces publicly available, we used communication patterns extracted from recent publications [2], [3], [38] to emulate the current network utilization in the datacenter (i.e., background traffic). The patterns used are described as follows:

- $Stride(i)$: a host with index x sends to the host with index $(x + i) mod(num\ hosts)$. We used the stride

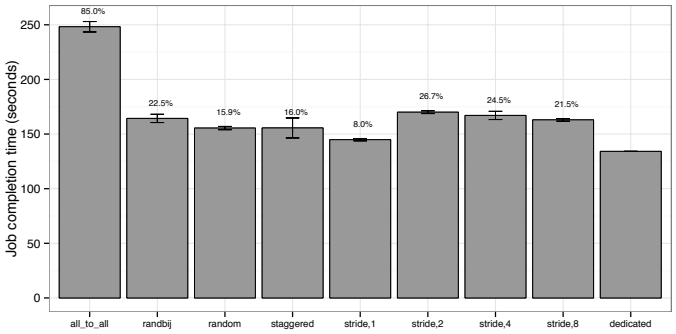

Fig. 7. Sort completion times with different background traffic patterns in a fat-Tree network.

pattern with i = 1, 2, 4, and 8. This traffic pattern emulates the case where traffic crosses all hierarchy layers of the network topology (edge, aggregation and core layers). This patterns is commonly induced by HPC applications [3].

- $Staggered(P_{ToR}, P_{Pod})$: a host sends to another host in the same ToR switch with probability P_{ToR}, to a host in the same POD with probability P_{Pod} and to the rest of the network with probability $1 - P_{ToR} - P_{Pod}$. We used this pattern with $P_{ToR} = 0.5$ and $P_{Pod} = 0.3$. This traffic pattern emulates the case where an application instance is deployed in a "proximity cluster" within a datacenter, thus confining most of data exchanges either in the same POD or even in the same rack [38].

- $Random$: a host sends to any other host in the network with a uniform probability. Hosts may receive from more than one host. $Randbij$ is a special case of the random pattern that performs bijective mapping.

- All_to_all: a host sends to all the other hosts in the network. This pattern emulates the case where a distributed application exchanges data with all hosts (e.g., data shuffling).

Figure 7 shows job completion times for the Sort MR application when co-existing with randomized, staggered, stride and all-to-all background traffic patterns, respectively. Times are reported in seconds and represent the average of multiple executions. It can be observed that application was impacted in virtually all the communication patterns we have tested. Moreover, we observe that performance impact is proportional to the network load imposed by the background traffic pattern. For example, patterns $stride4$ and $stride8$ generate significant load to the upper network layers. Although this heavily impacts job performance, it also increases the opportunity for traffic spreading since there are more path alternatives at the core layer. The pattern $stride2$ generates more traffic at the aggregation layer, where there fewer paths to choose from. For the pattern $stride1$, on the other hand, most of the traffic remains within the same Top-of-Rack switch and aggregate switches, which causes a less significant impact to MR application performance and provides less opportunity for optimization. The maximum impact was observed for the for the all_to_all pattern, where Sort was slowed down by 85%.

VI. Related Work

To the best of our knowledge, we are the first to present such an experimental framework. Past research and development activities related to our work falls within following categories.

Network emulation. FORCE [9] is perhaps the work most closely related. It allows for the creation of emulated datacenter networks on top of a small set of hardware OpenFlow-enabled switches and servers. Each server runs a number of virtual machines to emulate an entire datacenter rack and an Open vSwitch virtual switch to emulate the top-of-rack switch. It also provides a MapReduce-like traffic generator, as described below. Our tool differs from FORCE in that it does not require real network hardware. Instead, it relies on Mininet-HiFi to emulate an entire datacenter in a single server using lightweight virtualization. As described earlier, Mininet-HiFi can emulate arbitrary network topologies and has proven to be able to reproduce network research experiments with a high level of fidelity. Although we may lose scalability by running experiments in a single server, the next versions of Mininet-HiFi are expected to support distributed execution. Moreover, since our MapReduce emulation tool also works as a standalone tool, it could be used with FORCE, when the required hardware is available.

Network Traffic Generation. There are large number of network traffic generation tools/techniques available in the literature (a comprehensive list is provided by Botta et al. [39]). D-ITG [40] is one of the most prominent examples. It allows one to generate complex traffic patterns in datacenter networks. Although this type of tool can be configured to generate MapReduce-like traffic, using it to evaluate the impact of network research on MapReduce performance is not straightforward (e.g., impact in the job completion times). Recently, Moody et al. [9] presented a Hadoop shuffle traffic simulator, as part of the FORCE tool, that is supposed to place a realistic load on a datacenter network. However, the authors do not provide information about how transfers are scheduled and no comparison with real MapReduce traffic is performed. Thus, it is not possible to assess the accuracy of such a traffic generator in representing realistic MapReduce traffic. Our tool differs from past work in that it not only generates traffic in the same way a real MapReduce job would, but it also mimics the Hadoop internals that may be impacted by network performance. Thus, it is possible to use it to evaluate the impact of network research on MapReduce performance.

MapReduce Network Analysis. There are a number of studies on MapReduce performance, a few of them focusing on the network. Pythia [7] and Flowcomb [6] have shown that it is possible to accelerate MapReduce jobs by optimizing the network behavior against shuffle transfers. Wang et al. [31] used the MRPerf simulator, which relies on ns-2 to perform packet-level network simulation, to evaluate the impact of network topology on MapReduce application performance. They used four different topologies (non-blocking, dumbbell, single path tree and DCell) showing that the MapReduce shuffle phase perform better on topologies that provide higher aggregate bandwidth. Although it is not a goal of our work to provide a comprehensive characterization of network-related performance in MapReduce, our study goes further by performing fine-grained variation of available network bandwidth from

100Mbps to 10Gbps, including a case with partition skew, and evaluating the impact of background traffic to the application performance. The results suggest that it is possible to improve MapReduce performance by providing more bandwidth and that this improvement can be even greater in the presence of partition skew.

VII. Conclusions and Future Work

This paper presented MRemu, an emulation-based framework that enables conducting datacenter network research and resource planning, without requiring expensive and continuous access to large-scale datacenter hardware resources. Among the highlights of MRemu is a) the ability to process logs and traces from executions of real MapReduce applications in production datacenters and replay or extrapolate patterns to facilitate realistic network emulation in the context of datacenter network evaluation, b) the ability to emulate network control code for software-defined networks (SDN) implemented and running directly on the production SDN controller and c) the potential of its use as a standalone tool in legacy computer clusters with limited resources for real data-crunching. We showed through rigorous and methodical experimentation that MRemu exhibits high accuracy of emulation with respect to application performance, when compared to respective application runs in a real datacenter. As such, we are confident that MRemu can be used by the broader research community as a valuable tool in its experimentation toolset. In fact, we are working in this direction and towards making MRemu available to the community. As part of our next steps, we are working on overcoming the limitations mentioned in Section III-D (distributed emulation, multi-job execution), as well as incorporating further analytics runtimes beyond Hadoop MapReduce into the emulator.

References

[1] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing Data Transfers in Computer Clusters with Orchestra," in *Proceedings of the ACM SIGCOMM 2011 Conference*, 2011, pp. 98–109.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, Aug. 2008.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *Proceedings of the USENIX NSDI 2010 Conference*, 2010.

[4] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," *Commun. ACM*, vol. 54, no. 3, pp. 95–104, Mar. 2011.

[5] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-Performance Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 254–265, Aug. 2011.

[6] A. Das, C. Lumezanu, Y. Zhang, V. Singh, and G. Jiang, "Transparent and Flexible Network Management for Big Data Processing in the Cloud," in *5th USENIX Workshop on Hot Topics in Cloud Computing*, San Jose, CA, US, Jun. 2013.

[7] M. Neves, K. Katrinis, H. Franke, and C. De Rose, "Pythia: Faster Big Data in Motion through Predictive Software-Defined Network Optimization at Runtime," in *Proceedings of the IEEE IPDPS 2014 Conference.*, 2014.

[8] W. Cui and C. Qian, "DiFS: Distributed Flow Scheduling for Adaptive Routing in Hierarchical Data Center Networks," in *Proceedings of the ACM/IEEE ANCS 2014 Conference*, 2014.

[9] W. C. Moody, J. Anderson, K.-C. Wang, and A. Apon, "Reconfigurable Network Testbed for Evaluation of Datacenter Topologies," in *Proceedings of the ACM DIDC 2014 Conference*, 2014.

[10] A. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead Datacenter Traffic Management Using End-host-based Elephant Detection," in *Proceedings of the IEEE INFOCOM 2011 Conference*, Apr 2011, pp. 1629 –1637.

[11] M. Hammoud, M. S. Rehman, and M. F. Sakr, "Center-of-Gravity Reduce Task Scheduling to Lower MapReduce Network Traffic," in *Proceedings of the IEEE CLOUD 2012 Conference*, 2012, pp. 49–58.

[12] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible Network Experiments Using Container-Based Emulation," in *Proceedings of the ACM CoNEXT 2012 Conference*, 2012, pp. 253–264.

[13] Linux Foundation. (2015) OpenDaylight Project. Accessed on March 2015. [Online]. Available: http://www.opendaylight.org

[14] NOX/POX, "NOX/POX OpenFlow Controller," 2015, accessed on March 2015. [Online]. Available: http://www.noxrepo.org

[15] Apache Software Foundation, "Apache Hadoop Website," 2015, accessed on March 2015. [Online]. Available: http://hadoop.apache.org

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," in *Proceedings of the ACM SIGOPS/EuroSys 2007 Conference*, 2007.

[17] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A Runtime for Iterative MapReduce," in *Proceedings of the ACM HPDC 2010 Conference*, 2010, pp. 810–818.

[18] Apache Software Foundation, "Apache Spark," 2015, accessed on March 2015. [Online]. Available: http://spark-project.org.

[19] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[20] D. Borthakur, "The Hadoop Distributed File System: Architecture and Design," Jan 2007, accessed on March 2015. [Online]. Available: http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf

[21] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., Jun 2009.

[22] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The Case for Evaluating MapReduce Performance Using Workload Suites," in *Proceedings of the IEEE MASCOTS 2011 Conference*, 2011.

[23] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *Proceedings of the USENIX OSDI 2008*, 2008, pp. 29–42.

[24] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP Incast Throughput Collapse in Datacenter Networks," in *Proceedings of the ACM WREN 2009 Conference*, 2009, pp. 73–82.

[25] Apache Software Foundation, "Rumen," 2015, accessed on March 2015. [Online]. Available: http://goo.gl/8W4Riz

[26] JSON, "JavaScript Object Notation," 2015, accessed on March 2015. [Online]. Available: http://www.json.org

[27] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-defined Networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010.

[28] Mininet, "Reproducing Network Research," 2015, accessed on March 2015. [Online]. Available: http://goo.gl/idfDBR

[29] Mininet, "Mininet Cluster Edition," 2015, accessed on March 2015. [Online]. Available: http://goo.gl/Y6cl03

[30] Y. Peng, K. Chen, G. Wang, W. Bai, Z. Ma, and L. Gu, "HadoopWatch: A First Step Towards Comprehensive Traffic Forecasting in Cloud Computing," in *INFOCOM, 2014 Proceedings IEEE*, 2014, pp. 19–27.

[31] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, "A Simulation Approach to Evaluating Design Decisions in MapReduce Setups," in *Proceedings of the IEEE MASCOTS 2009 Conference*, 2009.

[32] Y. Liu, M. Li, N. K. Alham, and S. Hammoud, "HSim: A MapReduce simulator in enabling Cloud Computing," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 300–308, Jan. 2013.

[33] Intel Corporation, "Intel Distribution for Apache Hadoop Software: Optimization and Tuning Guide," 2012, accessed on March 2015. [Online]. Available: http://goo.gl/nf6AQ3

[34] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis," in *Proceedings of the IEEE ICDEW Conference*, 2010, pp. 41–51.

[35] Apache Software Foundation. (2015) Apatch Nutch. Accessed on March 2015. [Online]. Available: http://nutch.apache.org

[36] "PEGASUS: Peta-Scale Graph Mining System," 2015, accessed on March 2015. [Online]. Available: http://www.cs.cmu.edu/~pegasus/

[37] Apache Software Foundation, "Apache Mahout," 2015, accessed on March 2015. [Online]. Available: http://mahout.apache.org.

[38] X. Wu and X. Yang, "DARD: Distributed Adaptive Routing for Datacenter Networks," in *Proceedings of the IEEE ICDCS 2012 Conference*, 2012, pp. 32–41.

[39] A. Botta, A. Dainotti, and A. Pescapé, "A Tool for the Generation of Realistic Network Workload for Emerging Networking Scenarios," *Computer Networks*, vol. 56, no. 15, pp. 3531–3547, 2012.

[40] "D-ITG: Distributed Internet Traffic Generator," 2015, accessed on March 2015. [Online]. Available: http://traffic.comics.unina.it/software/ITG