Prophet: Scheduling Executors with Time-varying Resource Demands on Data-Parallel Computation Frameworks

Guoyao Xu*, Cheng-Zhong Xu*†, and Song Jiang*
*Department of Electrical and Computer Engineering, Wayne State University, Detroit, Michigan
†Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China
Email: {xu.yao, czxu, sjiang}@wayne.edu

***Abstract*—Efficiently scheduling execution instances of data-parallel computing frameworks, such as Spark and Dryad, on a multi-tenant environment is critical to applications' performance and systems' utilization. To this end, one has to avoid resource fragmentation and over-allocation so that both idleness and contention of resources can be minimized. To make effective scheduling decisions, a scheduler has to be informed of and exploit resource demands of individual execution instances, including both short-lived tasks and long-lived executors. The issue becomes particularly challenging when resource demands greatly vary over time within each instance. Prior studies often assume that a scheduling instance is either short lived or of gradually varying resource demands.**

However, when in-memory computing platforms, such as Spark, become increasingly popular, the assumption no longer holds. The execution instance for scheduling becomes executor, which executes an entire application once it is scheduled. Usually it is not short lived. Its resource demands are significantly time-varying. To address the inefficacy of current cluster schedulers, we propose a scheduling approach, namely *Prophet*, which takes resource demand variation within each executor into the scheduling decision. It leverages the fact that execution of a data-parallel application is pre-defined by a DAG structure and resource demands at various DAG stages are highly predictable. With this knowledge, Prophet schedules executors to minimize resource fragmentation and over-allocation. To deal with unexpected resource contention, Prophet adaptively backs off selected task(s) to reduce the contention. We have implemented Prophet in Apache Yarn running Spark. We evaluated it on a 16-server cluster, using 10 categories of a total of 90 application benchmarks. Compared to Yarn's default capacity and fair schedulers, Prophet reduces application makespan by up to 39% and reduces their median completion time by 23%.

I. Introduction

Applications running on today's large-scale data-parallel processing frameworks, such as Apache Hadoop [1], Dryad [2] and Spark [3], usually have DAG (Directed Acyclic Graph) composed of stages in their execution durations. Each stage consists of a number of tasks conducting the same type of data processing. A task requires multiple resources for its running, including CPU, memory, as well as disk and network bandwidths. While tasks belonging to the same stage are of similar demands for each of the resources, those belonging to different stages can have very different demands for different resources. For example, in machine learning applications, such as K-Means and SVM (Support Vector Machine) [4], [5], tasks of their map stages are I/O- and CPU-intensive and tasks of the reduce stages are network-intensive. The frameworks, such as Hadoop and Spark, usually run on a resource management system. YARN[6] and Mesos[7] are current popular systems that are responsible for resource allocation and sharing. It is critical for task schedulers in a YARN-like system to efficiently schedule the tasks of vastly diverse multi-resource demands onto a cluster of servers, so that both applications' execution time and the cluster's throughput can be maximized.

Scheduling tasks with multi-resource demands onto servers of limited amount of resources (CPU, memory, disk, and network) is often formulated as a multidimensional bin packing problem. As long as these demands are known a priori or can be accurately estimated, this problem can be solved heuristically in a polynomial time [8]. A common technique used for this estimation is to profiling tasks by leveraging the fact that jobs of an application are recurring and they "repeat hourly (or daily) to do the same computation on newly arriving data." [8].

Such a profiling strategy is not sufficient to fully address the issue by itself in practice, as the demands measured during a task's run vary (sometimes dramatically). While it is known that a multidimensional bin packing problem is NP-hard and has to be solved with heuristics, it is almost impossible to accommodate time-varying demands into the model to efficiently produce an effective scheduling decision. A conservative alternative is to use peak usage of a resource to represent the varying demands of a task during running to prevent resource *over-allocation* [8], which occurs when aggregate demand from all running tasks exceeds available resources. It often leads to interference between tasks and serious performance degradation. However, this conservative approach generates risk of resource *fragmentation*, which occurs when resources are idle but tasks with demands on them that ready for scheduling cannot use them.

To achieve high scheduling efficiency, a scheduler has to simultaneously minimize fragmentation and over-allocation of resources [8]. When each application can have a large number of tasks and each task has a relatively short execution time, using peak demand may not create extremely large pockets of fragmentation in terms of wasted resource time. However, this becomes a serious issue with in-memory computing frameworks, such as Spark [3] and Storm [9], where scheduling units have long execution time with varying demands.

An application of a Spark-like in-memory computing framework, does not expose its tasks to the underlying resource man-

agement system, like YARN and Mesos. Instead, the concept of executor[1] is introduced as the scheduling instance in these systems. Once executors of an application are launched on servers by the system's scheduler, the application's scheduler is responsible for scheduling its tasks to these pre-allocated executors. Specifically, an executor is usually a Java virtual machine (JVM) and tasks are threads running on the JVM. Each Spark application has a set of executors scheduled by the resource manager to different servers and they stay alive until all tasks of the application are completed. This two-level scheduling is adopted for two reasons. One is to cache a subset of data in memory to enable in-memory reuse of data across tasks in an executor in a fault-tolerant manner. The other is to significantly reduce overhead of launching tasks, which is critical for in-memory computing. In contrast, in a Hadoop application each task runs on a dedicated JVM, which is scheduled by the system's resource manager.

While there are two levels of scheduling for in-memory computing, the executors' scheduling plays a more performance-critical role as it represents the resources allocation and sharing between applications. Recent work like Tetris [8] exploits the knowledge of future (peak) resource demands of tasks. However, It cannot be applied directly on executors' scheduling. When an executor becomes the scheduling object, the rationale made by existing schedulers based on peak resource usage to represent the object's varying resource demand is less likely to be valid. An executor runs multiple batches of tasks belonging to different DAG stages may have (very) different resource demands. Therefore, using the peak demand to represent different demands of a resource during the lifetime of an executor for resource allocation can cause serious resource fragmentation (or wastage).

Additionally, for a smooth run of tasks in an executor without interference from other application executors, it might be desired to have all four major required resources (CPU, memory, disk, and network) pre-allocated or reserved. Users only need to pre-specify their resource demands on CPU (number of cores) and memory (size of memory) for an executor. As these demands usually represent the bottom line of a user's requirement on quality of service, the requested resources are pre-reserved at the time of executor scheduling. However, network and disk resources are shared among executors on a server without isolation or reservation. They are more likely to incur over-allocation, and tend to cause disk seeks or network incast that may significantly compromise system's throughput. In addition, neither users nor current cluster managers [6], [7] would specify network and disk demands of executors, let alone consider their highly variable demands. This may lead to application performance degradation and poor resource efficiency.

To improve cluster efficiency and speed up individual applications' performance for in-memory computation, we design an executor scheduler, namely *Prophet*, which can select an executor whose scheduling would result in the smallest amount of fragmentation and over-allocation of network and disk resources. With the knowledge of an executor's future varying (peak) disk and network demands at any stage during its lifetime and of each stage's start time and its duration, Prophet can estimate resource availability at any time frame in the near future and make an informed scheduling decision accordingly to minimize resource fragmentation and over-allocation. To deal with unexpected resource contention, Prophet selects task(s) in an executor to back off to adaptively ameliorate the contention.

In summary, we make the following contributions in the paper.

- We identify a performance-critical issue about the executor scheduling on in-memory data parallel computing platforms. We show that without considering resource demand variation within an executor, one can hardly enable an effective scheduling. By showing stability and predicability of resource demands in an executor, we make it possible to take the dynamics on the resource demands into account.
- We design an online executor scheduler, named *Prophet*, that adopts a greedy approach by choosing the currently optimal executors in terms of expected resource fragmentation and over-allocation to dispatch. It also dynamically avoids severe resource contention and subsequent dramatic performance degradation due to unexpected over-allocation with its task backoff mechanism.
- We have implemented Prophet on YARN and Spark 1.5 to support Spark and evaluated it on a 16-server cluster. Experiments show that Prophet can minimize resource fragmentation while avoiding over-allocation. It can substantially improve cluster resource utilization, minimize application makespan, and speed up application completion time. Compared to Yarn's default capacity and fair schedulers, Prophet reduces the makespan of workloads in SparkBench [4] by 39% and the median job completion time by 23%.

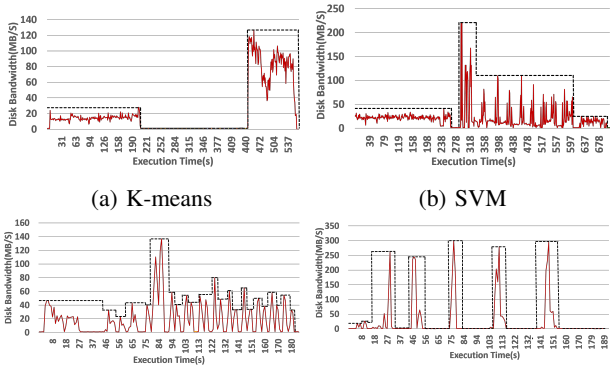
The rest of the paper is organized as follows. Section II describes motivation of the work and demonstrates predictability of resource demands in an executor. Section III describes design of the Prophet scheduling scheme. Section IV describes the implementation and evaluation of Prophet. Section V reviews the related work, and Section VI concludes the paper.

II. Motivation and Background

A. Workload Analysis

To illustrate the potential efficiency loss due to resource fragmentation and over-allocation, we use four Spark benchmarks and their input data generators available in SparkBench [4], to reveal their executors' resource demand variations. Among the four benchmarks, two (K-means and SVM) represent machine learning workloads, and the other two

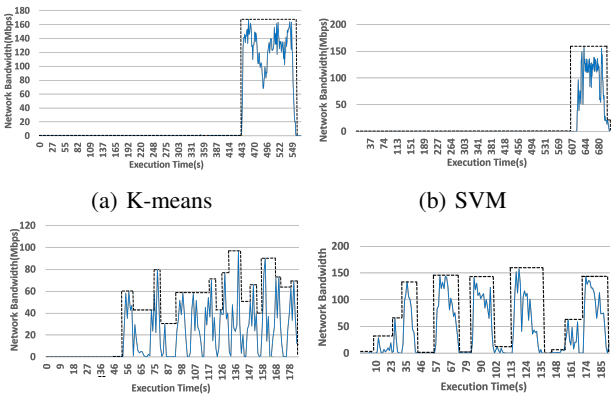
[1]The executor may be named differently. In the YARN environment, it is sometimes called container [6]. In the paper introducing Spark, it is called worker [3], while in the paper describing Mesos [10] and in Spark Apache's official website [11], it is called executor.



(a) K-means (b) SVM

(c) Pagerank (d) SVD++

Fig. 1: Disk bandwidth usages of four Spark benchmarks (K-means, SVM, PageRank, and SVD++). DAG stages are marked with dotted lines.



(a) K-means (b) SVM

(c) Pagerank (d) SVD++

Fig. 2: Network bandwidth usages of four Spark benchmarks (K-means, SVM, PageRank, and SVD++). DAG stages are marked with dotted lines.

(PageRank and SVD++) represent graph computation workloads. They are briefly described in the below.

- K-means is a machine learning workload clustering a set of data into K clusters.
- SVM (Support Vector Machine), is a machine learning classifier workload analyzing data and recognizing patterns of high dimensional feature spaces while efficiently conducting non-linear classifications.
- PageRank is a graph computation workload ranking website pages and estimating their importance.
- SVD++ is a graph computation collaborative filtering workload improving the quality of recommendation system based on the users' feedbacks.

Figures 1 and 2 show disk and network bandwidth demands of the four Spark benchmarks (Spark 1.5.0) on Hadoop Yarn 2.4.0, respectively. Each executor is exclusively run on a server of 24 cores, 32GB of memory, three 7200 RPM disk drives, and 1Gbps NIC. It is obvious that for both disk and network usages the amount of requested bandwidth varies from almost 0 MB/s to around 300MB/s for disk or around 160MB/s for network. Their very low resource demands can stay for more than half of some executors' lifetimes, such as for network usages of K-means and SVM, while their peak demands are still very high, such as around 160MB/s. Should the resources be allocated according to the peak demands, they would be significantly wasted due to the serious fragmentation. Even worse, starvation may occur on applications with both high peak network and disk demands as servers may not have available resources to meet both peak demands simultaneously (even though such an availability is not necessary). On the other hand, if they are not pre-allocated, multiple executors on the same server may simultaneously experience high demand on the same resource, causing resource over-allocation. This can lead to severe interference (disk seeks or network incast) between the executors, which can sharply degrade applications' performance.

It is necessary to take resource variation of executors into their scheduling decision so that both resource fragmentation and over-allocation can be minimized. This is a highly challenging issue considering that even scheduling objects of constant resource demands (e.g., using peak demands) can be NP-hard [8].

B. Predictability

Recent studies on large-scale data-parallel systems reveal that most applications in production clusters exhibit recurring execution behaviors with predictable future resource demands and mostly constant execution time in each DAG stage for given CPU cores and with sufficient memory [8], [12], [13], [14], [15]. Therefore, tasks' statistics measured in their prior runs enable effective estimation. Specifically, "since tasks in a phase perform the same computation on different partitions of data, their resource use is statistically similar." [8]. An offline or online profiling of tasks' runs would provide a scheduler with knowledge on tasks' resource demands. To illustrate this, in addition to the aforementioned four benchmarks, we select another six Spark benchmarks. Three of them (LR, TriangleCount, and TeraSort) are from SparkBench [4], and the other three (WordCount, Sort, and Grep) are from BigDataBench [16]. They are described in the below.

- Logistic Regression (LR) is a machine learning classifier benchmark to predict continuous or categorical data.
- TriangleCount is a fundamental graph analytics counting number of triangles in a graph to detect spam or hidden structures in web pages.
- TeraSort is a sorting benchmark using map/reduce to sort input data into a total order.
- WordCount reads Wikipedia text entries as input, and counts how often words occur.
- Sort is a benchmark designed for sorting words from a Wikipedia dataset.
- Grep is a benchmark filtering and finding specified words from a Wikipedia dataset.

For each of the ten benchmarks, we used 9 settings, including three CPU core numbers for each executor (one, three,

TABLE I: Three categories of input dataset sizes for each of 10 benchmarks

Benchmarks	SVM	KMeans	LR	PageRank	SVD++	TriangleCount	Terasort	WordCount	Sort	Grep
Large Input Dataset	38.3G	21.9G	37.1G	4.0G	365.6M	364.7M	37.3G	44G	44G	44G
Medium Input Dataset	19.2G	10.9G	18.5G	1.9G	163.3M	167.2M	18.6G	22G	22G	22G
Small Input Dataset	9.6G	5.5G	9.3G	933.1M	78.1M	86.5M	9.3G	11G	11G	11G

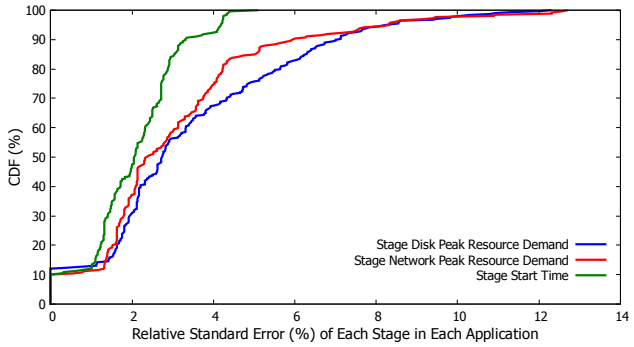


Fig. 3: Relative standard errors of disk/network bandwidth and stage start time over the five runs of each of 10 benchmarks with different settings on CPU core and input size. Each run uses a different input dataset.

and five) and three categories of input dataset sizes (small, medium, and large). The dataset sizes for each benchmark and category are shown in Table I. Each of the settings run five times with different input datasets of the same size. For each of the five runs in a dedicated cluster of 16 nodes, we collect each stage's start time and peak disk/netowrk bandwidths of an executor and compute their relative standard errors over the five runs. Figure 3 plots the errors with CDF (cumulative distribution function) curves. As shown, the relative errors are mostly smaller than 10%. Though contents of the input data sets have the potential of affecting executor's behaviors, such as number of iterations to reach a convergence in machine learning applications, the impact is small. More importantly, each stage's start time is very stable (with a 5% or smaller relative standard error).

Because usually the same setting (CPU cores for each executor and input dataset size) remains in use for an application for an extended time period [14], [15], [13], profiling results about stage start time and peak resource demands of a run is sufficient for an executor scheduler to make an informed decision for its future runs. However, when an application uses a new setting that has not been profiled, we need a method to estimate the results. To this end, we adopt a supported vector machine (SVM) with linear regression technique. In particular, we feed results from 25 profiling runs covering representative settings into the machine to build a prediction model. The model then takes in a new setting (about CPU cores and dataset size) and produces its predicted stage start time and peak resource demands. Because changing CPU core count and input size usually does not lead to disruptive change of an executor's behaviors, the model consistently provides high-quality estimations (mostly less than 10% errors).

III. Design of Prophet

As an executor scheduler, in addition to its main objective of minimizing resource fragmentation and over-allocation, Prophet has two other objectives. One is fairness across applications, and the other is load balance across servers running applications. In the scheduling, all arriving applications will be placed into a waiting queue. When an application is submitted, its required reserved CPU, memory, and number of executors are specified by users. When there exist applications whose specified CPU and memory resource demands can be met by currently available resources in the cluster, Prophet greedily chooses one that would result in minimal fragmentation and over-allocation of network and disk resources for dispatching. Then the required number of executors are created on different servers. Note that for load balance across servers in an application's execution, Prophet always creates the required number of executors at the time when the application is scheduled. It does not create executors fewer than the required ones when resources are not sufficient. Otherwise, if the number of executors is allowed to increase, all newly created executors will request data from existing ones and cause them to become performance bottleneck. For fairness and starvation avoidance, Prophet chooses an application for scheduling from a subset of pending applications that have waited for the longest time (by default 50% of all pending ones). Each application is also assigned a deadline when it arrives at the queue. It will be scheduled immediately when its deadline is passed. The deadline can be assigned according to current average waiting time (e.g. three times of its average).

A. Prophet's Scheduling Algorithm

Prophet's scheduling algorithm is designed under the assumption that future resource demands of an executor, either one that is running or one that is candidate to be scheduled, is known in advance (or can be predicted). By knowing the total demands of executors currently running at a server, Prophet can compute how much the resource would be available in the near future. This is illustrated in Figure 4 for disk bandwidth of a server with two executors being scheduled on it. In the figure, each executor has two stages of different peak disk bandwidth demands. However, their combined effect leaves the available resource of four distinct values, or four *resource availability stages*. At this time we have two candidate applications' executors for Prophet to decide which one to schedule, as shown in Figures 5(a) and (b), respectively.

If only disk bandwidth is considered, Prophet needs to examine future fragmentation areas (FAs) and over-allocated areas (OAs) in Figure 5. FA or OA refers to the area between the two lines for available bandwidth and the demand in the

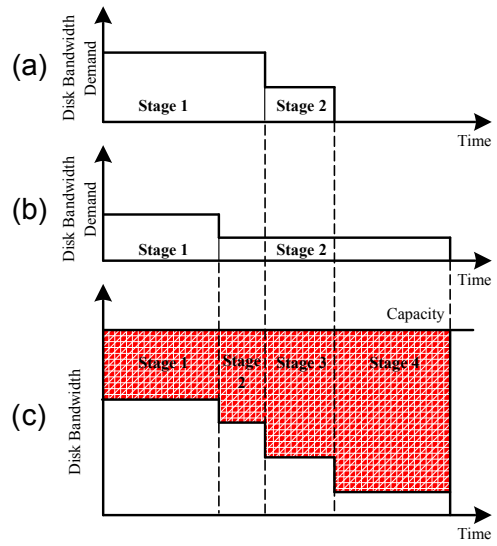


Fig. 4: Illustration of predicting available disk bandwidth. With known peak demands on disk bandwidth across stages of two executors (see (a) and (b)), the shaded area in (c) between their combined demand and the disk's capacity represents the disk's bandwidth to be available.

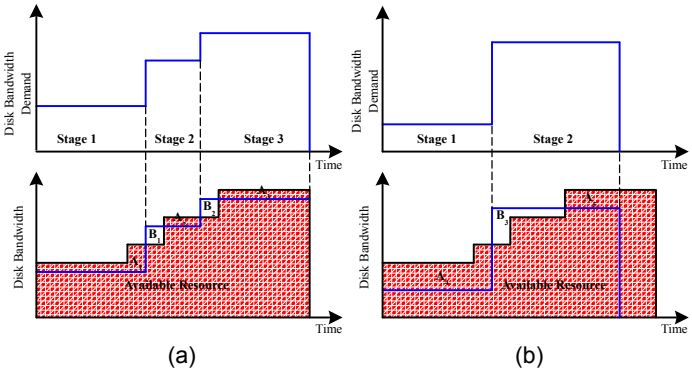


Fig. 5: Illustration of how fragmentation area (FA) and over-allocation area (OA) of disk bandwidth are formed for two executors. For each executor (see (a) or (b)), the graph at the top shows its peak demands on disk bandwidth across stages, and the graph at the bottom shows the demand and available disk resource (shaded area computed in Figure 4) overlap with each other to form FAs, such as A_1, A_2, and A_3, and OAs, such as B_1 and B_2.

figure. If available bandwidth is larger than the demand, it is FA, such as $A_i(i = 1, 2, ...5)$. Otherwise, it is OA, such as $B_i, (i = 1, 2, 3)$. FA represents wasted resource and OA suggests resource contention and performance degradation. A good scheduler would simultaneously minimize the two areas. In this example, Prophet will schedule the executor shown in Figure 5(a), as it has much smaller aggregate FA/OA area than that in Figure 5(b). This example also indicates a scheduler that is unaware of future resource demands and availability might schedule the executor shown in Figure 5(b), leading to much worse performance.

To formally describe the design of the scheduling algorithm,

TABLE II: Notations in the Prophet's scheduling algorithm, r indicates network or disk resource

C_i^r	Capacity of Resource r on Server i
$P_k^{r,j}$	Peak demand of Resource r from Executor j at its Stage k
$A_s^{r,i}$	Available Resource r of Server i at resource Stage s
$t_k^{j,start}$,$t_k^{j,end}$	Start and end times of Stage k at Executor j
$T_s^{i,start}$,$T_s^{i,end}$	Start and end times of resource Stage s at Server i

we introduce a number of notations as shown in Table II. Note that in the notations, quantities about duration and times ($t_k^{i,start}$, $t_k^{i,end}$, $T_s^{i,start}$, and $T_s^{i,end}$) are not defined specifically for certain resource. Instead, they are specified according to change of stages for any resources.

To quantify fragmentation and over-allocation for candidate application's executors, we may simply add FA or OA of an executor's every stage, and consider the sum as the executor's fragmentation score or over-allocation score, or F and O in short, respectively. However, for an executor of many stages, prediction on demands and resource availabilities at the earlier stages, or those closer to the current time, is usually more accurate than that on later stages, because the latter is more likely to be influenced by unaccounted noises. For this reason, we give earlier stages a higher weight. Specifically, if the executor has n stages, the weight for Stage i ($i = 0, 1, ..., n-1$) is $w_i = 1 - i/n$. Therefore, the two scores can be computed for Resource r as following.

For any $P_k^{r,j} \leq A_s^{r,i}$,

$$F_r = \sum_k \left(\sum_s Formula(s,k) \right); \tag{1}$$

For any $P_k^{r,j} > A_s^{r,i}$,

$$O_r = \sum_k \left(\sum_s Formula(s,k) \right); \tag{2}$$

If $t_k^{j,end} \leq T_s^{i,start}$ or $T_s^{i,end} \leq t_k^{j,start}$,

$$Formula(s,k) = 0 \tag{3}$$

else if $t_k^{j,start} \leq T_s^{i,start} \leq T_s^{i,end} \leq t_k^{j,end}$,

$$Formula(s,k) = \left[|P_k^{r,j} - A_s^{r,i}| * \left(T_s^{i,end} - T_s^{i,start} \right) \right] * w_k \tag{4}$$

else if $t_k^{j,start} \leq T_s^{i,start} \leq t_k^{j,end} \leq T_s^{i,end}$,

$$Formula(s,k) = \left[|P_k^{r,j} - A_s^{r,i}| * \left(t_k^{j,end} - T_s^{i,start} \right) \right] * w_k \tag{5}$$

else if $T_s^{i,start} \leq t_k^{j,start} \leq T_s^{i,end} \leq t_k^{j,end}$,

$$Formula(s,k) = \left[|P_k^{r,j} - A_s^{r,i}| * \left(T_s^{i,end} - t_k^{j,start} \right) \right] * w_k \tag{6}$$

else if $T_s^{i,start} \leq t_k^{j,start} \leq t_k^{j,end} \leq T_s^{i,end}$,

$$Formula(s,k) = \left[|P_k^{r,j} - A_s^{r,i}| * \left(t_k^{j,end} - t_k^{j,start} \right) \right] * w_k. \quad (7)$$

In theory, to minimize both fragmentation and over-allocation in the selection of applications for scheduling, we might simply use the sum of the two scores as the metric for the selection. However, resource over-allocation can cause contention among executors and slow down all involved ones. More seriously, the slowdown may lead to more idleness (fragmentation) of other resources. To address the issue, we give O a higher weight when computing the overall score.

$$OverallScore_r = (1-\eta)O_r + \eta * F_r. \quad (8)$$

In our prototype, we set η as 0.3 by default, which is experimentally determined to balance the risks of severe performance degradation and wastage of resources. We leave a comprehensive study of this parameter as future work.

While for each resource (disk or network resources) Prophet can compute an overall score, for all resources it obtains a vector of overall scores for an application's executor. To convert the vector into a one-dimensional quantity for comparison across candidate applications, we use the Euclidean norm of the vector. Accordingly Prophet selects an application whose executors have the smallest norm. The scheduling algorithm is described in Algorithm 1.

Algorithm 1 Prophet Scheduling Algorithm

Denote the Available Resource of Server i as: AR_i
Denote the User Pre-Defined CPU and Memory Resource Requirement of Executor j as: RR_j
Denote the Overall Score Vector of Executor j as: OSV_j
When Executor j of application p is added to queue
Offline Predictor predicts its $P_k^{r,j}$,$t_k^{j,start}$,$t_k^{j,end}$
When a hearbeat is received from Server i
while there is $AR_i\{cpu, memory\}$ on Server i **do**
  for each Executor j in the queue **do**
    if $RR_j\{cpu, memory\} < AR_i\{cpu, memory\}$ **then**
      Acquire latest predicted $P_k^{r,j}$,$A_s^{r,i}$ from Predictor
      Compute OSV_j of Executor j
    else
      OSV_j = NULL
    end if
  end for
  Launch Executor j whose Euclidean norm of OSV_j is minimum and not empty on Server i
  Update $AR_i\{cpu, memory\}$, $A_s^{r,i}$ of Server i
end while

B. Ameliorating Contention with Task Backoff

While Prophet attempts to avoid expected over-allocations, there still can be unexpected ones or expected minor ones turn out to be major over-allocations. As we have indicated in Section I, severe over-allocation leads to intensive interference. For disk and network, such an interference can cause their effective bandwidths to be much lower than their normal peak ones due to reasons such as random access and incast, respectively. When interference essentially blocks tasks of an executor from moving forward, the executor's reserved CPU cores and memory are also wasted. To address the issue, Prophet has an exception handling mechanism built in the Spark's task scheduler. When it is observed that effective disk or network bandwidth is substantially lower than their peak one but the disk or network stays busy to serve requests at a server, a serious over-allocation is detected at the server. Prophet will examine the profiled resource demands of each executor on the server and identify ones that are most likely to overuse the contested resource. It then activates a backoff mechanism by reducing number of tasks dispatched to the executors until the effective bandwidth approaches the peak one or the resource is not busy anymore. Note that the mechanism is enabled only temporarily, usually lasting for only a few task scheduling rounds, as an overaction could compromise utilization of CPU and memory.

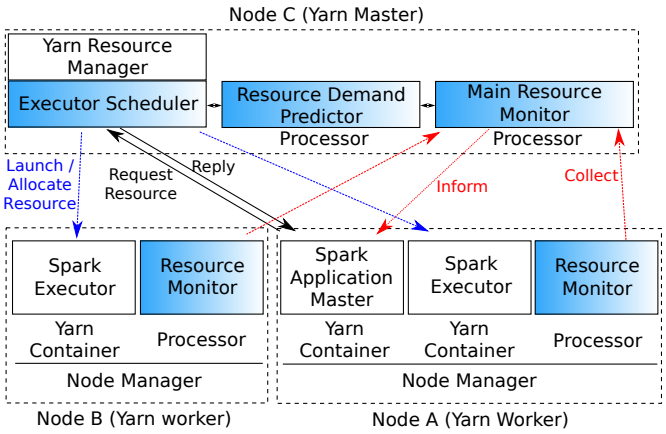


Fig. 6: The framework of Spark applications running on a Yarn cluster, in which Prophet modules are included (shown as shaded boxes).

IV. Implementation and Evaluation of Prophet

We implemented Prophet executor scheduler on Hadoop YARN 2.4.0 and the task backoff mechanism in Spark 1.5.0. In addition, we implemented a resource usage monitor on each server to detect over-allocation. In this section, we provide implementation details, system settings for performance evaluation, and evaluation results.

A. Prophet's Implementation

Figure 6 depicts where the Prophet modules are situated in the framework of Spark applications running on a YARN cluster. Yarn's cluster-wide resource manager is responsible for receiving executors' resource requests from each Spark application master, and communicating with node manager at each server to decide whether there is sufficient amount of

resource to meet the resource demands. If yes, corresponding resources will be allocated, and the Spark application master and its executors will run as containers on servers managed by node manager. On this framework we made a few instrumentations.

- The resource demand predictor runs as a separate process on the Yarn's master node hosting its resource manager. In the background it continuously learns and predicts executors' resource demands.
- The executor scheduler is enabled as Yarn's plug-in scheduler. It communicates with the predictor before making its scheduling decisions.
- The task backoff mechanism is implemented in Spark's scheduler, which runs with each Spark application master and communicates with the resource monitor to decide whether task backoff should be enabled for an application and if yes, for how long.
- The main resource monitor running as a separate background process at the master node communicates and collects information from the resource monitors running at worker nodes.

These changes are lightweight. They minimally increase complexity and scalability of Yarn's scheduling framework. The profiling and prediction are running in the background.

B. Experiment Setup

We deployed our implementation of Prophet in Hadoop Yarn 2.4.0 and Spark 1.5.0 on a 16-server cluster. Each server has 24 cores, 32GB of memory, three 3.5TB 7200 RPM disk drives with a 110MB/s peak bandwidth for each one. It has a 1Gbps NIC and runs Linux 3.16. We used the 10 benchmarks that were described in Section 1. In the same as we ran the benchmarks in Section 1, for each benchmark, we varied its input size as listed in Table I and its CPU core count (1, 3, or 5). So essentially we had 90 applications to run in the evaluation. Each application was submitted to the system at a time randomly picked between 0 second (experiment start time) and 1200 seconds.

The input datasets of the machine learning and graph computation benchmarks (K-means, SVM, Pagerank, SVD++, LR, and TriangleCount) are kept in memory as Spark RDD abstraction to support following parameter vector calculation, update, and broadcast in each iteration.

We compared Prophet to three state-of-the-art Spark scheduling algorithms implemented in Yarn, which are Dominant Resource Fairness(DRF) scheduler [17], the capacity scheduler(CS) [18], [19], and Tetris [8]. The capacity scheduler was designed to achieve fairness on memory allocation based on Hadoop's slot-based resource management, while DRF considers fairness for both CPU and memory. In addition to CPU and memory, Tetris considers network and disk bandwidths. It tries to efficiently pack tasks/executors when available resources are sufficient to accommodate their peak demands. Nevertheless, to the best of our knowledge, all the existing schedulers were designed for task-grained scheduling without considering resource demand variations during execution of a scheduling object.

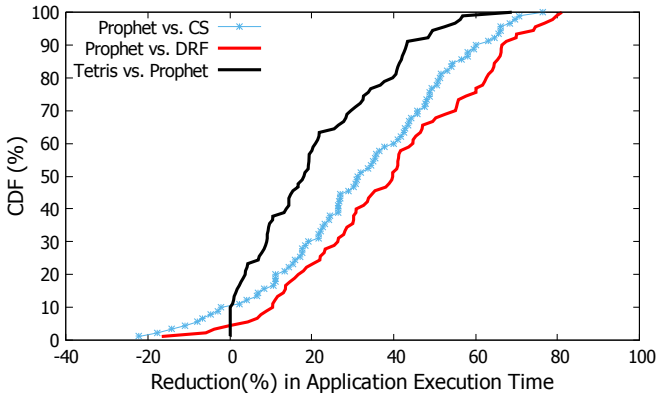


Fig. 7: CDF curves for reductions of execution times by Scheduler X over Scheduler Y, shown as X vs. Y. X can be Prophet and Tetris, and Y can be CS, DRF, and Prophet.

C. Experiment Results

Figure 7 shows cumulative distribution function (CDF) curves about reduction of application's execution time by Prophet over CS, by Prophet over DRF, and by Tetris over Prophet. An application's execution time is measured from the time its executors are scheduled to its completion. The figure shows substantial gaps on execution time between the schedulers. For example, the figure shows that there are 50% of applications whose execution times are reduced by up to 31% if they are scheduled by Prophet over those by CS, reduced by up to 40% by Prophet over those by DRF, or by up to 18% by Tetris over those by Prophet.

While CS considers only memory and DRF considers only memory and CPU, it is a surprise to see Prophet generally performs better than them in terms of execution time. Prophet uses prediction and task backoff to avoid over-allocation of disk and network bandwidths. In contrast, CS and DRF experience (much) more serious interference between executors at a server, and take a longer time period to complete. However, it is interesting to have these two observations. First, there are a few percentage of applications whose CS/DRF execution time is shorter than that of Prophet. This is because Prophet also manages to reduce fragmentation, which may increase risks of interference. In such cases, CS and DRF may win. Second, Tetris consistently has a shorter execution time than Prophet. Execution time can only be compromised by over-allocation, and not by fragmentation. Tetris uses an executor's peak resource demands for allocation. So it is less likely to have an over-allocation. However, Prophet also needs to consider reducing fragmentation, which does not help with execution time. As we know, a metric more meaningful to users is completion time, which is defined as the period from an application's submission time to its completion time.

Figure 8 shows CDF curves of application's completion time reduction by Prophet over CS, by Prophet over DRF, and by Prophet over Tetris. For this metric, Prophet is better

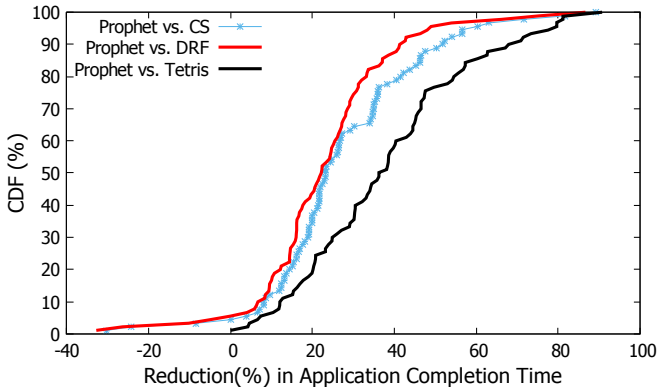


Fig. 8: CDF curves for reductions of completion times by Scheduler X over Scheduler Y, shown as X vs. Y. X is Prophet, and Y can be CS, DRF, and Tetris.

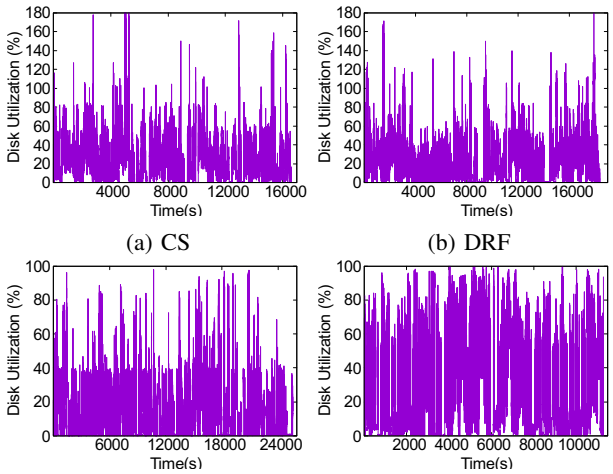
TABLE III: Makespans produced by various schedulers for running the 90 applications

Scheduler	CS	DRF	Tetris	Prophet	Propeht w/o Backoff
Makespan (s)	16604	18369	25537	11290	15707

than Tetris. For example, there are 50% of applications whose completion time is reduced by 36% or more, and 10% of applications whose time is reduced by 12% or less. If we read makespans of the executions under different schedulers listed in Table III, it is clear that Prophet is much better than other schedulers. A makespan measures total time period used to complete all the 90 applications under a particular scheduler. It is directly correlated to the system's resource efficiency. Prophet reduces the makespan by 32%, 39%, and 56% compared to CS, DRF, and Tetris, respectively. The reduction over Tetris is the most significant, while Tetris produces the best application execution time. It is because Prophet has substantially reduced the waiting time of applications by improving resource efficiency, while moderately compromising the execution time.

These results reveal the strength of Prophet, which is aware of varying future resource demands and takes them into scheduling decision. If a scheduler does not have the knowledge, it has two options. One option, that is taken by Tetris, conservatively uses executors' peak demands for scheduling. While this minimizes possibility of over-allocations and helps with the execution time, it would leave significant fragmentations, which compromises resource efficiency. Therefore, it is expected to see that Tetris has the worst makespan. Another option, that is taken CS and DRF, simply does not consider disk and network demands in their scheduling. So they are more likely to have serious interference than Tetris and Prophet. That is why their execution time is worse. In the meantime, they are less likely to have fragmentations than Teris. That is why their makespans are shorter than Tetris. By explicitly considering varying resource demands, Prophet can address both over-allocation and fragmentation issues.

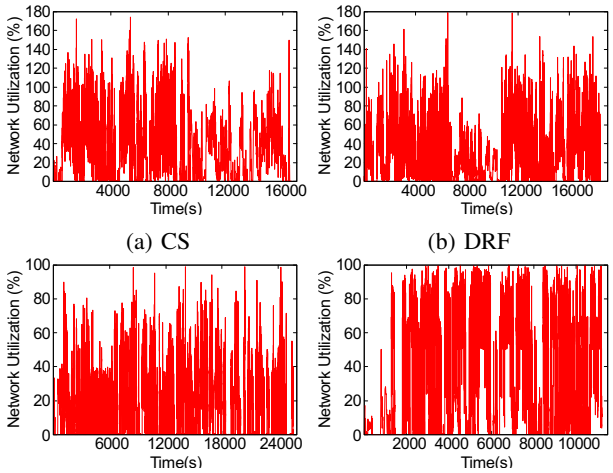
To reveal insights on how disk and network bandwidths are actually consumed, we show their utilizations under the four



(a) CS (b) DRF

(c) Tetris (d) Prophet

Fig. 9: Disk utilizations during running 90 applications under various schedulers.



(a) CS (b) DRF

(c) Tetris (d) Prophet

Fig. 10: Network utilizations during running 90 applications under various schedulers.

schedulers in Figures 9 and 10. The utilization is the ratio between aggregate demand on a resource from all executors at a server and the server's capacity of that resource. As shown, for CS and DRF, there are many significant over-allocations, which suggest much lower effective (disk or network) bandwidth. In contrast, Tetris and Prophet have little over-allocation. However, there are much higher utilization values in Prophet than those in Tetris (for either disk or network) , indicating that Tetris has much more serious fragmentation issue.

While Prophet has two components to achieve its scheduling objectives, we would like to see the contribution made by each of the components (prediction-based scheduling policy and task backoff mechanism). Figure 11 shows CDF curves for reductions of execution time and completion time by Prophet over a Prophet version without a task backoff mechanism.

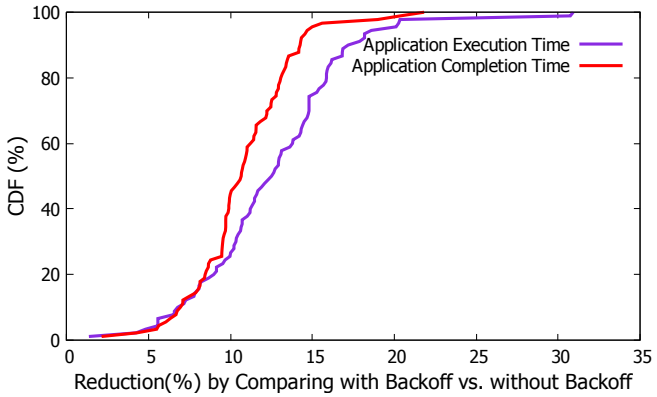


Fig. 11: CDF curves for reductions of execution and completion times by Prophet over Prophet without task backoff mechanism.

While the prediction-based scheduling policy tries to minimize both fragmentation and over-allocation, the task backoff mechanism basically addresses only the over-allocation issue. Application execution time is directly affected by interference caused by over-allocation. As Figure 11 shows, without the backoff mechanism, applications' median completion and execution time suffers 10% and 13% degradation, respectively, when unexpected over-allocation occurs. It is often caused by the absence of adequate predictability. Simultaneously, it implicitly demonstrates the impact of predictability on the scheduling quality. This experiment also reveals that in a shared execution platform, it is necessary to have a backup mechanism to keep the system from unexpectedly high resource demands.

V. Related Work

There have been many studies on task scheduling on the data-parallel computing platforms. There are also studies on leveraging history resource usage to predict future resource demands for improving data locality and execution efficiency. In addition, studies on virtual machine placement and migration are also related on the aspect that Spark's executors are actually Java virtual machines. In the below we show how Prophet scheduling is related to these works and why it represents a unique contribution.

Shared Use of Network/Disk : In current cluster management systems, network and disk resources are usually shared among tasks or executors without pre-reservation and isolation [6], [7]. While such a sharing approach helps to reduce fragmentations, it increases the risk of resource contention and interference. To address the issue, Tetris uses peak demands on network and disk in its task scheduling scheme to determine how many and what tasks can be placed on a machine of limited resources [8]. To address the same issue, Quasar and Paragon adopt task classification techniques, and adjust resource scale-up and scale-out allocations based on their calculated constant demands to avoid over-allocation [20], [21]. In addition, they try to co-locate tasks of complementary network and disk demands for higher resource utilization. However, these schemes can potentially produce significant fragmentations when scheduling executors of Spark applications, where each executor has multiple stages with (highly) varying resource demands during its extended execution time period. The key reason is that their assumption on stable resource demands in a task does not hold any longer. In contrast, Prophet considers varying demands in executors' scheduling to address both fragmentation and over-allocation issues.

Cluster Schedulers: The issue of task scheduling in large-scale data-parallel systems has been extensively studied over past few years [22], [2], [3]. Quincy and delay schedulers are designed to improve data locality of individual task while maintaining fairness of different applications [23], [24]. Both Fair and Capacity Scheduler [18], [19] are Yarn's default schedulers [6] designed for slot-based resource allocation. They dispatch tasks for high scalability and fairness. Dominant Resource Fairness (DRF) utilizes max-min fairness to maximize the minimum dominant share for all users when allocating multiple resources [17]. Implementations of DRF or earlier schedulers only consider CPU and memory in their resource allocations. The ILA scheduler is proposed as an interference- and locality-aware scheduling strategy for MapReduce virtual clusters [25]. However, none of existing works are designed for scheduling Spark executors, where Prophet is.

Plan-ahead Scheduler Techniques: It is known that future resource demands and execution times are predictable in recurring applications, which are common in current data analytic production clusters. There are optimization techniques that take advantage of this knowledge to plan ahead and make scheduling decisions accordingly. Apollo estimates task execution time from historical task runtime statistics to perform estimation-based task delay scheduling for improved data locality and reduced task completion time [12]. Tetris estimates tasks' peak multi-resource demands from previous runtime statistics to conduct multi-dimensional task-packing scheduling [8]. Both Jockey and ARIA predict the completion time of a running application through past execution profiles and a control loop estimating applications progress, to automatically adjust resource allocation to meet applications SLO [13], [26]. Corral predicts future application arrival time and characteristics, such as input and intermediate data sizes, from recurring application statistics. It then coordinates input data placement with task placement to improve data locality and reduce amount of cross-rack network data transfer [14]. Although prediction on application behaviors based on its history has been shown to be highly effective for informed scheduling, none of these approaches could be applied for scheduling of Spark's executors. While in-memory computing platforms, including Spark, become popular, the efficient scheduling based on prediction is on high demand, and Prophet makes a timely contribution.

VM Packing/Schedulers: To some extent, virtual machine (VM) scheduling bears much resemblance to executor scheduling. Both need to consider demands of multiple resources and

both will host multiple processes or tasks to run. Because the actual resource demands of VMs and executors can be highly variable, their scheduling need to avoid both resource over-allocation and fragmentation. There have a number of works on VM scheduling. Among them, AutoControl automatically adapts resource allocation to dynamic workload changes based on feedback control approach [27]. Sandpiper migrates VM to alleviate overload condition to maximize resource utilization [28]. However, they do not adopt predictive plan-ahead packing placement strategy to avoid interference due to workloads co-location because future resource demands of VMs running Web-based or interactive applications are highly unpredictable. Hence neither of these approaches could be applied to scheduling of Spark executors, because it would be too expensive to (frequently) migrate data-intensive executors and incur transferring of large volume of data.

VI. Conclusion

Existing task schedulers are not suitable for scheduling executors with time-varying resource demands on an in-memory data-parallel computing platform, such as Spark. They suffer from serious over-allocation and fragmentation problems and can substantially compromise application performance and system resource utilization. Motivated by observations on recurring resource usage patterns in the platform, we propose a scheduling scheme, Prophet, to learn and leverage the patterns in the executors' scheduling. In particular, Prophet can accurately predict resource availability at a server and varying demands from executors in the near future. It allows the demands to best match available resources. This will help with both application performance and system utilization. In addition, Prophet has a task backoff mechanism to accommodate unexpected over-allocation to improve the system's robustness.

We have implemented Prophet on Yarn and Spark. Extensive experiments with publicly available benchmarks show that Prophet can reduce makespan by up to 39% and median application completion time by 23%, compared to Yarns default capacity and fair schedulers.

Acknowledgment

The authors would like to thank the reviewers for their constructive comments. In particular, Dr. Chenyang Lu, as shepherd of the paper, provided valuable suggestions to improve its quality. We also thank Ruiqi Sun and Xingbo Wu for their comments and discussions. This work was supported in part by the China National Basic Research Program (973 Program, No. 2015CB352400), NSFC under Grant U1401258 and Research Program of Shenzhen JSGG20150512145714248. This work was also partially supported by US National Science Foundation under Grants 1217948 and 1527076.

References

[1] "Apache hadoop," http://hadoop.apache.org.
[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *SIGOPS*, 2007.
[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.
[4] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark," in *CF*, 2015.
[5] "Spark benchmark suite," https://github.com/SparkTC/spark-bench.
[6] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *SOCC*, 2013.
[7] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *NSDI*, 2011.
[8] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *SIGCOMM*, 2014.
[9] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *SIGMOD*, 2014.
[10] "Apache mesos," http://mesos.apache.org/.
[11] "Apache spark," http://spark.apache.org/.
[12] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *OSDI*, 2014.
[13] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: guaranteed job latency in data parallel clusters," in *Eurosys*, 2012.
[14] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," in *SIGCOMM*, 2015.
[15] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *NSDI*, 2012.
[16] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: a big data benchmark suite from internet services," in *HPCA*, 2014.
[17] A.Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *NSDI*, 2011.
[18] "Hadoop mapreduce - fair scheduler." http://bit.ly/1p7sJ1I.
[19] "Hadoop mapreduce - capacity scheduler." http://bit.ly/1tGpbDN.
[20] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," in *ASPLOS*, 2014.
[21] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ASPLOS*, 2013.
[22] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, 2008.
[23] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *SOSP*, 2009.
[24] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Eurosys*, 2010.
[25] X. Bu, J. Rao, and C.-z. Xu, "Interference and locality-aware task scheduling for mapreduce applications in virtual clusters," in *HPDC*, 2013.
[26] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," in *ICAC*, 2011.
[27] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated control of multiple virtualized resources," in *Eurosys*, 2009.
[28] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif, "Black-box and gray-box strategies for virtual machine migration." in *NSDI*, 2007.