Elasecutor: Elastic Executor Scheduling in Data Analytics Systems

Libin Liu
City University of Hong Kong
Hong Kong
libinliu-c@my.cityu.edu.hk

Hong Xu
City University of Hong Kong
Hong Kong
henry.xu@cityu.edu.hk

ABSTRACT

Modern data analytics systems use long-running executors to run an application's entire DAG. Executors exhibit salient time-varying resource requirements. Yet, existing schedulers simply reserve resources for executors statically, and use the peak resource demand to guide executor placement. This leads to low utilization and poor application performance.

We present Elasecutor, a novel executor scheduler for data analytics systems. Elasecutor dynamically allocates and explicitly sizes resources to executors over time according to the predicted time-varying resource demands. Rather than placing executors using their peak demand, Elasecutor strategically assigns them to machines based on a concept called *dominant remaining resource* to minimize resource fragmentation. Elasecutor further adaptively reprovisions resources in order to tolerate inaccurate demand prediction. Testbed evaluation on a 35-node cluster with our Spark-based prototype implementation shows that Elasecutor reduces makespan by more than 42% on average, reduces median application completion time by up to 40%, and improves cluster utilization by up to 55% compared to existing work.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; • **Software and its engineering** → **Scheduling**; *Cloud computing*; • **Theory of computation** → *Approximation algorithms analysis*;

KEYWORDS

Data analytics systems, elastic scheduling, executor

ACM Reference Format:
Libin Liu and Hong Xu. 2018. Elasecutor: Elastic Executor Scheduling in Data Analytics Systems. In *Proceedings of SoCC'18: ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC'18),* 14 pages. https://doi.org/10.1145/3267809.3267818

1 INTRODUCTION

Data analytics systems are widely used to process big data [1–3, 5, 22, 34, 37, 39, 50, 65, 77, 79]. The workflow of an analytics application can be expressed as a DAG (Directed Acyclic Graph), which is composed of different stages of processing. Each stage runs a number of tasks on worker machines, and each task performs the same computation on different partitions of data [20, 22, 31, 32, 48, 52, 54, 55, 79].

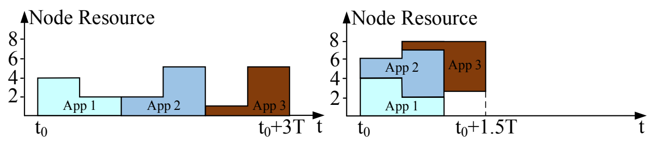
Resource scheduling is critical in data analytics systems. Many resource schedulers have been developed for various objectives, such as fairness, cluster utilization, application completion time, etc. [19, 24, 26, 27, 27, 31, 35, 38, 40, 46, 49, 56, 60, 66, 67, 69, 80]. Most are developed for a task-based execution model. They assume a task, which corresponds to one stage of the application DAG and is an individual process, is the basic execution unit and therefore the basic scheduling unit. This holds in general for batch processing systems like Hadoop [2] and cluster schedulers such as Yarn [67]. In Yarn for instance, tasks run in individual "containers" where Java virtual machines (JVMs) are spawned. A "container" runs one task only and is shut down after the taskfinishes.

However, in-memory analytics systems such as Spark [3] and Storm [4] rely on a different executor-based model [44, 71]. An executor is a long-running JVM process that executes an application's entire DAG [7, 79]. Once an executor is launched, the scheduler can dispatch different tasks to it. This enables data reuse across tasks and significantly reduces the overhead of launching tasks which is critical for fast in-memory processing [55, 79]. Executor-based systems usually adopt task-based resource schedulers for simplicity. This, however, leads to various performance and efficiency problems as a result of the unfitting assumptions.

First, since the resource usage of a task is roughly constant, most schedulers use static allocation [19, 21, 29, 31, 35, 49, 56, 57, 60, 67]. However, executors naturally exhibit time-varying resource usages since they run the application's entire DAG. As we will show in §2, an executor's peak-to-trough resource usage can be as high as 409x for extended periods of time. Existing schedulers then have to use peak demands to reserve resources [21, 31, 35, 67] which leads to cluster underutilization and degraded makespan[1]. Applications may also have to wait longer for enough resources to become available, resulting in prolonged completion times and poor user experience. Figure 1 illustrates this using a toy example.

Recent work such as Morpheus [66] addresses this by dynamically reserving resources and adjusting the number of executors as the application progresses. Also, some work [29, 31–33, 40] tries to increase task parallelism within an executor. That is, we keep

[1]Makespan measures the total time used to complete all applications.



(a) Static allocation **(b) Elastic allocation**

Figure 1: A toy example to demonstrate the drawbacks of static allocation. The host's capacity is 8, and applications are submitted at time t_0 and their resource demand time series are shown. In (a), static allocation runs all three applications sequentially, since their peak demands prevent them from running concurrently. In (b), elastic allocation allows them to run as long as their demand time series can be packed together, and reduces the makespan by 50%.

the executor resource allocationfi xed, and increase or decrease the number of tasks run in parallel within the executors. Both methods are shown to improve resource utilization. However, they do not fundamentally address the problem, because each executor is still allocated afi xed bundle of resources during the entire time, which still incurs underutilization or overallocation of different resources in face of time-varying multi-resource demands across different stages.

Besides resource allocation, a scheduler also needs to carefully assign executors to machines in a cluster. Usually this is done using the executor's peak demands and machine's remaining capacity [35, 67, 69]. Such an approach does not precisely capture the time-series utilization of the worker machines and executors and creates fragmentation, when resources are idle but cannot be used to schedule executors that are ready. We give two examples of fragmentation: (1) An application's peak demand may exceed a machine's remaining capacity at the moment, and it cannot be placed on this machine. Yet, it is possible that the peak demand only happens in a later time, at which point the machine will have enough capacity to run it (because some applications will have finished by then); (2) An application with a very short period of high peak demand is selected to runfi rst, preventing applications with stable demand from being scheduled on the same machine.

To address these problems, we propose Elasecutor, a novel executor scheduler that exploits time-varying resource demands for resource allocation and executor assignment. Elasecutor considers multiple resources: CPU, memory, network, and disk I/O. It exploits the recurring nature of many applications in production [15, 17, 31, 49, 66], and predicts the resource demands of applications over its lifetime. Elasecutor elastically allocates resources to executors according to the predicted time series of demands in order to reduce underutilization. It then packs executors strategically onto machines to minimize fragmentation among multiple resources and improve makespan.

We make the following contributions in this paper.

- We make a case for elastic executor scheduling (§2). We show through measurements with real workloads that Spark executors exhibit significant time-varying resource usage patterns (§2.1). We further experimentally establish the predictability of executor's demand time series (§2.2).
- We design a new scheduler called Elasecutor (§3), that allocates time-varying resources to executors and assigns them to machines based on the predicted demand time series (§3.2). To do so, Elasecutor relies on a concept called *dominant remaining resource* to search for executors whose demand time series best matches the time series of a machine's available resources. Elasecutor also reprovisions resource dynamically at runtime to compensate for possible prediction errors (§3.3).
- We implement Elasecutor on Spark (§4) and present a realistic performance evaluation on a 35-machine cluster (§5). Experiments with real workloads show that Elasecutor substantially improves performance. Compared to existing solutions such as Tetris [31], Elasecutor reduces makespan by up to 63%, reduces the median application completion time by up to 40%, and improves resource utilization by up to 55%.

2 MOTIVATION

We motivate our work by highlighting the limitations of peak based static executor allocation and assignment commonly used in current systems (§2.1). We also show that the executor resource usage can be fairly well predicted when the workloads are recurring (§2.2).

2.1 Need for an Elastic Scheduler

Workloads	Terasort	K-means	LR	PageRank
Dataset Size	32.0GB	37.4GB	37.3GB	2.8GB

Table 1: Input dataset size for profiling the four Spark workloads from the HiBench bigdata benchmark suite [10]. Each workload runs with 20 executors, each using at most 3 cores and 8GB memory.

Executors are the basic scheduling unit in Spark and similar systems. Each application has dedicated executors to run its tasks [7]. Current executor schedulers [3, 21, 35, 67] work in virtually the same way as task schedulers for systems like Hadoop [2]. Users need to specify the resource demands of an executor, so the scheduler can make resource reservations. The resources allocated to an executor are static and released only after the applicationfi nishes. Thus peak resource demands have to be used for allocation.

We argue that current schedulers can lead to severe resource underutilization because application resource usage varies greatly in different stages of data processing [31, 54, 66, 74, 75]. To demonstrate this, we profile the resource usage of executor processes using several typical Spark workloads as shown in Table 1, including Terasort, K-means, Logistic Regression (LR), and Pagerank. These workloads are commonly used in existing work [10, 33, 36, 54, 55]. We develop a monitoring module on Spark to collect CPU, memory, network I/O, and disk I/O usage of an executor. The measurements are done on our 35-machine testbed described in §5.1.

From Figure 2, we observe that the executor resource usage exhibits significant temporal variations. CPU usage varies all the way from only 4% to 100% for all applications. Similarly, memory usage ranges from 500 MB to nearly 6.8 GB for K-means. The network I/O varies from almost 0 Gbps to ∼4 Gbps for LR, and disk I/O from almost 0 MB/s to ∼190 MB/s for Terasort, respectively.

Applications	Terasort				K-means				LR				Pagerank			
	CPU	Memory	Net	Disk	CPU	Memory	Net	Disk	CPU	Memory	Net	Disk	CPU	Memory	Net	Disk
P/A	1.8	1.7	6.2	1.5	1.7	1.2	11.5	5.6	2.1	1.4	5.5	6.1	3.9	1.3	20.2	9.1
P/T	60	3.3	237	6.1	75	6	53	100	50	12.0	409.6	42.5	50	11.5	119	50
DP/T (%)	22.4	1.5	1.5	10.7	15.4	3.9	1.9	1.9	15.0	3.7	0.6	6.8	1.5	21.8	0.2	0.8
DHP/T (%)	36.5	67	10.3	73.2	51.9	90.4	5.8	13.5	40.4	91.0	19.5	11.3	23	84	2.5	7.1

Table 2: Statistical analysis of the time-varying resource usage of executors for the four Spark workloads (P/A = Peak Usage/Average Usage, P/T = Peak Usage/Trough Usage, DP/T = Duration of Peak Usage/Total Runtime, DHP/T = Duration for exceeding Half of Peak Usage/Total Runtime).

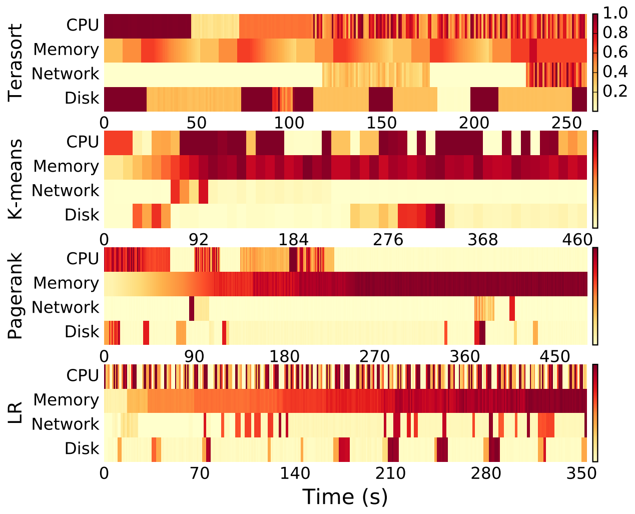


Figure 2: Heat maps of resource usages for four Spark workloads. The resource usages are normalized to the highest value of the executor. For all applications, the highest CPU usage is 3 cores; Terasort, K-means, Pagerank, and LR have 5.9GB, 6.8GB, 6.1GB, and 6.5GB highest memory usage; 2.4Gbps, 0.53Gbps, 0.62Gbps, and 4Gbps highest network usage; and 181MB/s, 103MB/s, 96MB/s, 170MB/s highest disk I/O usage.

More than half of the time an executor actually uses very little resources. Table 2 further shows our detailed analysis for executor's resource usage. We can see that the peak-to-trough ratio is high, ranging from 3.3 to 409.6. The period of peak resource usage takes up at most 22.4% of total runtime, and more than half of the runtime the resource usage actually falls below 50% of the peak except for memory.

Therefore static allocation using peak demands would clearly cause severe resource wastage and performance issues.

Some recent work [35, 66] has considered dynamically adjusting the number of executors according to the workload in order to improve utilization. Yet each executor still gets afi xed bundle of resources (CPU and memory) during the application's entire runtime. As observed in Figure 2, the usages of different resources do not correlate strongly. Thus such an approach does not fundamentally solve the over-allocation issue. Moreover, they only consider CPU and memory, and lack control over shared network and disk I/O resources. This results in possibly severe contention of shared resources which may then lead to underutilization of other resources.

2.2 Predictability of Resource Usage Time Series

To design an elastic executor scheduler, we need to have prior information about the time series of resource demands for executors. We now show that such information can be fairly easily predicted in practice.

Recent studies report that many production analytics workloads are recurring, such as running the same queries periodically on new data [15, 17, 31, 49, 66]. Further, the running times of the workloads are mostly constant given the same amount of input data and resources [15, 26, 31, 52, 66, 76]. Hence we can predict an executor's future resource requirements based on profiling its previous runs.

To see this, we measure eight workloads each with three input dataset sizes shown in Table 3. We vary the number of CPU cores from one tofi ve and memory from 2GB to 10GB correspondingly [11], creatingfi ve different resource profiles. Then for each application (a workload with certain dataset size and CPU and memory setting), we run a resource profilefi ve times, each time with different datasets of the same size. Other settings are the same as in §2.1. We collect the completion times of each stage and executor's peak resource usage in each stage, then calculate the coefficient of variation (CoV) over thefi ve runs. Figure 3 shows the cumulative distribution of CoVs for per-stage execution time and per-stage peak resource usage. To make it clearer, we also use Table 4 to show the corresponding statistical analysis of CoVs. We can see that CoVs are in general smaller than 14% and each stage's execution time is quite stable (90%ile CoV is less than 5.5%), as a CoV value less than 1 is considered to be low variance [26].

Therefore, for most recurring workloads it is accurate enough to use the profiling results from previous runs with same resource setting to represent the resource demands. For workloads with new settings, we can also build a prediction model [17, 23, 26, 68, 75] to infer their time-series resource usage, which we detail in §4. Note that for new applications, we need to collect data for several runs before we can predict the resource demand time series. Again given that most workloads are recurring, we believe this does not affect the usability of our system.

Workloads	Sort	WordCount	Terasort	Bayes	K-means	LR	PageRank	NWeight
Dataset 1	3.1G	3.1G	3.2G	1.8G	3.7G	7.5G	1.7M	37.5M
Dataset 2	30.6G	30.6G	32G	3.5G	18.7G	22.4G	247.9M	294.5M
Dataset 3	286.8G	305.9G	320G	70.1G	37.4G	37.3G	2.8G	2.7G

Table 3: Three types of input dataset sizes for the eight workloads of the HiBench bigdata benchmark suite [10].

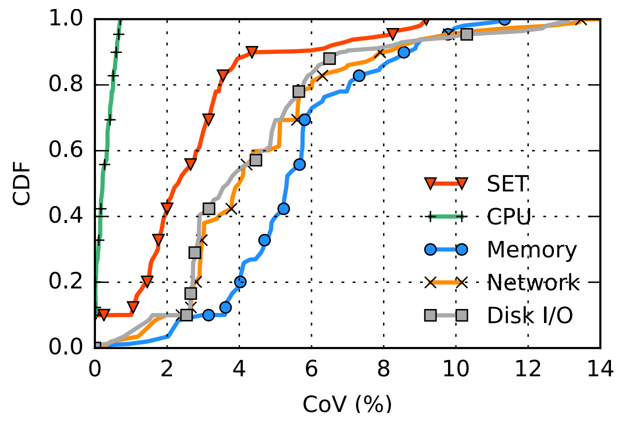


Figure 3: The CDFs of coefficient of variations (CoVs) for per-stage execution time (SET) and resource peak usage of each executor overfi ve runs for each of the eight workloads with different settings on CPU core and input data size shown in Table 3. Each run uses a different input dataset.

CoV Statistics (%)	Percentiles			
	10th	50th	90th	99th
SET	0.7	2.6	5.5	9.1
CPU	0	0.3	0.6	0.7
Memory	3.1	5.6	8.6	11.0
Network	2.4	4.2	7.9	13.4
Disk I/O	2.5	2.9	6.8	12.9

Table 4: Statistical analysis of CoVs.

3 DESIGN

We now present the design of Elasecutor in this section. We start by presenting the system overview in §3.1. We then explain in detail the elastic executor scheduling algorithm in §3.2, which is the core contribution of the design. Lastly, we discuss in §3.3 how Elasecutor uses dynamic reprovisioning at runtime to minimize the impact of prediction errors in inferring the executor's resource demand.

3.1 Overview

Elasecutor is an executor resource scheduler for data analytics systems. It predicts executors' time-varying resource demands (step 1 in Figure 4), collects workers' available resources (step 2), assigns executors to machines to minimize fragmentation (steps 3 and 4), elastically allocates resources (step 5), and leverages dynamic reprovisioning for better application QoS (steps 6 and 7).

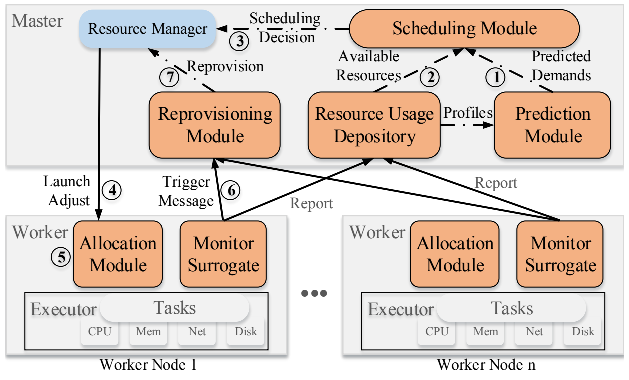


Figure 4: Elasecutor overview (key components are highlighted).

We explain several key components here.

Monitor Surrogate. Elasecutor employs a *monitor surrogate* at each worker node to continuously monitor the resource usage of executors in real-time. It collects the process-level CPU, memory, network I/O, and disk I/O usage, and reports the time series profiles to the resource usage depository (RUD) at the master node via RPC. The information is then used to build machine learning models to predict executor resource time series. The monitor surrogate also reports the node's future available resources to the RUD. Moreover, it monitors executor progress to see whether reprovisioning should be triggered due to significant prediction errors.

Resource Usage Depository (RUD). The RUD runs as a background process at the master node communicating with monitor surrogates and collecting information at each heartbeat of 3s. For simplicity we use a single master node and one RUD process, which is sufficient in our testbed evaluation. We can scale the RUD to multiple cores or multiple masters for large-scale deployment following many similar designs in distributed control plane [43], which is beyond the scope of this paper.

Scheduling Module. The scheduling module decides how resources should be allocated to executors and which executors should be assigned to machines. It obtains an application's demand time series from the prediction module which we will introduce shortly. It then packs executors to machines across multiple resource types, in order to avoid overallocation and minimize fragmentation throughout the executor's lifetime. For this purpose, we design a scheduling algorithm based on a novel metric called *dominant remaining resource (DRR)* which is detailed in §3.2. Once a scheduling decision is made, the selected worker IDs along with the executor IDs are sent to Spark's resource manager [7], which instructs the corresponding workers to launch the executors.

Allocation Module. This module explicitly and dynamically sizes the resource bundles to the executor process according to the resource manager's instructions. Through this, Elasecutor implements elastic allocation based on time-varying demands, which is illustrated in detail in §4.

Reprovisioning Module. Dynamic reprovisioning mainly deals with cases when the executor's actual resource usage deviates significantly from the predicted time series, which is unavoidable in practice. When an executor's progress is detected by the monitor surrogate to be slower than expected, the reprovisioning module is activated to calculate extra resources needed to make up for the slowdown. The corresponding algorithm is discussed in §3.3.

Prediction Module. Finally, the prediction module runs as a background process at the master node. It continuously fetches executor resource profiles from the RUD to train a prediction model for application's resource demand time series. Many machine learning and time series analysis techniques can be used for this purpose, which is not the focus of this paper. §4 provides more information about the prediction algorithm we currently use.

3.2 Elastic Executor Scheduling

The foremost challenge Elasecutor faces is how to elastically schedule executors with their multi-resource demand time series. We explain our solution to this challenge here.

We focus on recurring applications which are common in production settings [17, 31, 49, 66, 68]. When an application request is submitted, it specifies the number of executors and their configurations. Elasecutor predicts the per-executor resource demand time series based on such information and the past runs of this application. Elasecutor strives to satisfy the application's request completely, instead of scaling up or down the number of executors based on some fairness criteria. This is because users value performance consistency or predictability much more than fairness in practice [66]. In cases when resources are insufficient applications will simply wait.

3.2.1 *An Analytic Model.*

We begin with an analytic model to capture the problem.

In our model, we consider four resources: CPU, memory, network I/O, and disk I/O. For each resource r, we denote its capacity on machine j as C_j^r. The per-executor demand of application i on resource r is $D_i^r(t')$ when it is running at t' into its lifetime. Once started, application i runs for T_i time slots, and the number of executors required is N_i. All these are known to the scheduler. Let $x_i(t)$ be the decision variable for scheduling, i.e. $x_i(t) = 1$ if application i is running at time slot t, and let t_i denote the time when application i starts. Let y_{ij} be the decision variable for executor assignment. That is, y_{ij} indicates the number of executors assigned on machine j for application i.

Constraints: First, we assume that applications cannot be paused or preempted once scheduled, which is consistent with prior work [31, 32, 66]. Thus,

$$x_i(t) = \begin{cases} 1, & t_i \le t \le t_i + T_i, \\ 0, & \text{otherwise}, \end{cases} \quad \forall i. \tag{1}$$

Second, the cumulative resource usage on a machine at any given time t cannot exceed its capacity. Each executor's resource allocation is exactly equal to the predicted demand $D_i^r(t-t_i)$ for resource r when it has been running since t_i (without interruption). Thus,

$$\sum_i x_i(t) y_{ij} D_i^r(t-t_i) \le C_j^r, \ \forall r, t, j. \tag{2}$$

The scheduler always allocates exactly N_i executors for application i as stated in the beginning of §3.2.

$$\sum_j y_{ij} = N_i, \ \forall i. \tag{3}$$

Objective Function and Analysis. The objective of the scheduling algorithm is to minimize the makespan across all applications. Under a schedule $\{t_i\}$, application i finishes at $t_i + T_i$, and the makespan is $\max_i(t_i + T_i)$. Thus the scheduling problem can be formulated as the following:

$$\begin{aligned} \min_{t_i} \quad & \max_i (t_i + T_i) \\ \text{s.t.} \quad & (1), (2), (3). \end{aligned} \tag{4}$$

It is also possible to use other objective functions, such as application completion time, in our formulation.

Finding an optimal schedule to the above problem (4) is difficult. The objective function and constraints (2) are nonlinear, which makes the problem computationally expensive to solve. Inspite of ignoring the objective function and the time-varying nature of resource demand, the problem of packing multi-dimensional balls (executors) to minimum number of bins (machines) is APX-Hard [64, 72]. Moreover, what we have here is an offline setting. The online version where applications arrive dynamically is even more difficult to solve with reasonable competitive ratio. Therefore, most prior work for the packing problems relies on heuristics.

Clearly, makespan would be minimized if all available resources along time could be utilized by applications seamlessly. Naturally, the basis for minimizing makespan is to avoid resource underutilization and minimize machine-level resource fragmentation. Therefore, Elasecutor aims to schedule executors to the bestfi tted machine in order to minimize multi-resource fragmentation.

We now introduce our heuristic scheduling algorithm.

3.2.2 *Packing Executors with MinFrag.*

A well-known heuristic to one-dimensional packing problem is Best Fit Decreasing (BFD). BFD proceeds by repeatedly matching the largest ball that canfit in the current bin until no more ballsfi t, then open a new bin. Intuitively, this approach reduces fragmentation and thus the number of bins used. BFD requires no more than $\frac{11}{9}OPT + 1$ bins, where OPT is the optimal number of bins [28].

Our heuristic *MinFrag* extends BFD, by transforming the multi-dimensional (and time-varying) bin packing problem into the classic one-dimensional problem. Such a transformation relies on a metric called *dominant remaining resource (DRR)*, which we illustratefirst.

DRR. The DRR of a machine is defined similar to dominant resource of a job in [29]. For a given machine, wefi nd the earliest time t at which an executor, among those running on this machine, willfinish according to the predicted demand time series. We then compute machine j's average remaining resource over time for the period up to t, which can be denoted as $R(*, j, t)/C(*, j, t)$ for resource

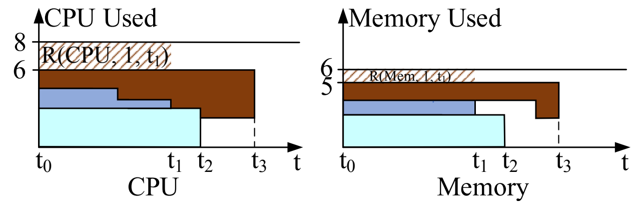


Figure 5: An illustration example for DRR. We just use two kinds of resources as an example. Three executors running on machine 1 consume CPU and memory variously over their lifetime. They all start to run at time t_0 and stop at time t_1, t_2, and t_3, respectively, and we assume machine 1's CPU and memory capacity are 8 and 6. Elasecutor calculates DRR at t_1, which is 1/4 in this case.

Algorithm 1 *MinFrag* pseudo code

```
1: when a heartbeat received from machine $j$
2: update $AR(j)$
3: while there are pending executors and $AR(j) > 0$ do
4:     for each pending executor $i$ do
5:         if $RD(i) < AR(j)$ then
6:             compute $\theta(i, j)$
7:     select $i^* = \arg\min_i \theta(i, j)$
8:     launch executor $i^*$ on $j$
9:     update $AR(j)$ and $\theta(j)$
```

Notation	Explanation
$AR(j)$	Time-series of future available resource at machine j
$RD(i)$	Time-series of resource demands of executor i
$\theta(j)$	DRR of machine j
$\theta(i, j)$	Updated DRR of machine j when executor i is placed on it

Table 5: Notations in *MinFrag*.

$*$. $R(*, j, t)$ is the integral of the amount of remaining resource along the time dimension up to t, and $C(*, j, t)$ is the integral of total capacity up to t. The machine j's DRR is then the maximum remaining resource among all types.

Figure 5 shows an example. We select t_1 as the time point to calculate DRR for machine 1. $\frac{R(\text{CPU},1,t_1)}{C(\text{CPU},1,t_1)} = 1/4$ and $\frac{R(\text{Mem},1,t_1)}{C(\text{Mem},1,t_1)} = 1/6$, and its DRR is simply 1/4.

Note we use the maximum remaining resource, not the minimum, because it better reflects machine utilization. If a machine's maximum remaining resource is 10%, then utilization of all resources is at least 90%. However if a machine's minimum remaining resource is 10%, utilization of some resources may still be lower than 90%. Minimum remaining resource reflects a machine's ability or potential to run executors, which is important if our objective is to minimize machine utilization.

MinFrag. As explained, *MinFrag* is based on BFD. On a high level, *MinFrag* works by iteratively assigning the "largest" executor to a machine that yields the *minimum* DRR in order to maximize utilization and improve makespan. We illustrate it in detail now.

Algorithm 1 shows *MinFrag*, and Table 5 lists the notations used here. When a heartbeat is received from a machine j, *MinFrag*

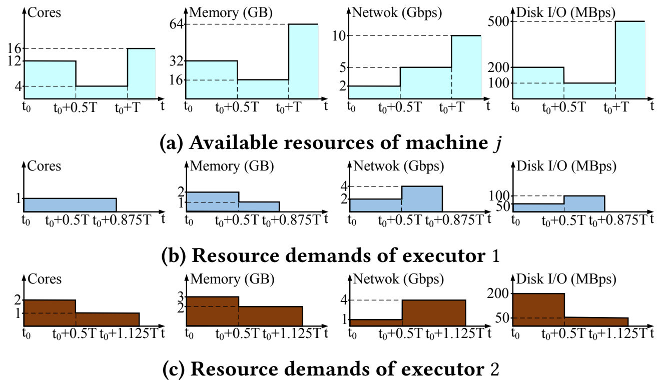


Figure 6: An illustration example of *MinFrag*. (a) The remaining resources of machine j until time $t_0 + T$, when a running executor will complete its execution. (b) The time-series resource usage of executor 1 which is expected tofinish at time $t_0 + 0.875T$. (c) The time-series resource usage of executor 2 which is expected tofi nish at time $t_0 + 1.125T$. The capacities of machines are: 16 cores for CPU, 64 GB for memory, 10 Gbps for network, and 500MB/s for disk I/O.

updates its available resources $AR(j)$ and then repeatedly does the following. It identifies if there is any executors that can run with enough resources on the machine. If yes, it computes the machine's DRR $\theta(i, j)$ if the executor was placed on it. Then among all eligible executors, *MinFrag* chooses i^* that minimizes $\theta(i, j)$, i.e. the largest executor. It updates the placement result, the machine's DRR $\theta(j)$, and available resource $AR(j)$. It then repeats the process until all executors are scheduled or there are no more available resources on the machine for any pending executor to run.

We use an example in Figure 6 to illustrate how *MinFrag* works. Figure 6a shows the machine's remaining capacity up to time T, when a running executor will complete its execution. There are two executors to schedule, and their demand time series are shown in Figures 6b and 6c, respectively. If executor 1 is assigned to the machine, $t_0 + 0.875T$ is the time point for calculating DRR, and the DRR $\theta(1, j) = \max\{\frac{53}{112}, \frac{165}{448}, \frac{3}{70}, \frac{6}{35}\} = \frac{53}{112}$. If executor 2 is assigned to the machine, $t_0 + T$ is the time point for calculating DRR, and the DRR $\theta(2, j) = \max\{\frac{13}{32}, \frac{43}{128}, \frac{1}{10}, \frac{1}{20}\} = \frac{13}{32}$. *MinFrag* then schedules executor 2 to run which minimizes DRR and thus maximizes utilization in this case. After taking executor 2, this machine does not have any network bandwidth and thus cannot take any more executors at the moment.

Some may argue that we can compute a score for each resource based on its remaining capacity, and convert the vector into a scalar value for comparison across candidate executors (say based on the Euclidean norm). However, the values for different resources have different units which makes such comparison irrelevant. Considering the remaining resource at its face value does not faithfully represent the degree of fragmentation as machines may have different capacities. The ratio between remaining resource and its capacity can represent actual fragmentation. Our evaluation results in §5.5 corroborate our argument.

3.3 Dynamic Reprovisioning

It is difficult to perfectly predict the time-varying resource demands due to many exogenous factors, such as infrastructure issues (e.g., hardware replacements, driver updates, etc.) and application issues (changes in size and skew of input data, changes in code/functionalities, etc.). Thus we design a dynamic reprovisioning mechanism that adjusts resource allocation online to tolerate prediction errors in Elasecutor.

Reprovisioning is triggered when an executor's progress is longer than ρ times the expected execution time of a processing stage of the application's DAG. We set ρ to 1.1, which is experimentally determined to balance application performance and resource wastage.

Given reprovisioning is required, we wish tofi nd out the actual resource demand of the application now. Elasecutor does so by temporarily allocating all remaining resource of the machine to this executor for one monitoring period (3s), and observe the executor's resource usage in this period. Suppose the actual usage of the executor i is $u(i, *)$ during this period for resource $*$, and the predicted demand is $p(i, *)$. Elasecutor then scales the allocation proportional to $u(i, *)/p(i, *)$ across all resources for all the remaining processing stages of the executor, and returns the remaining resources to the machine. This heuristic allows Elasecutor to quickly correct resource allocation without causing many missed scheduling opportunities at the machine.

When the machine has little remaining resources, the executor's actual usage observed in the reprovisioning period may be obscured and do not reflect its true demand. This is one limitation of our heuristic. Nonetheless, our evaluation results in §5.5 show that reprovisioning does improve application performance.

4 IMPLEMENTATION

We implement a prototype of Elasecutor with ~1K LOC in Python, Java, and Scala based on Spark 2.1.0. We open source our prototype here [8]. The monitor surrogate and allocation module at each worker machine communicate with the master node via RPC. We use lightweight system-level tools such as `psutil` and `jvmtop` in Linux to implement the monitor surrogate. We use `resourceRef`, which is a data structure that includes worker ID, application ID, executor ID, and corresponding resource time series. The scheduling module implements the *MinFrag* algorithm and dispatches the scheduling decisions to Spark's resource manager, which launches executors on worker machines. We modify the `launchExecutor()` function at the resource manager to send the predicted resource demand time series to the corresponding workers, along with other information. The allocation module then uses the modified `cgroups` and `OpenJDK` [13] to configure resources of the executor process based on the prediction results.

Allocation Module. Once a worker node receives the message for launching executors or resizing them from the resource manager, the allocation module adopts subsystems of `cgroups` to configure the time period and the limits of CPU, network, and disk I/O the executor process is entitled to. However, for Java-based systems, the maximum heap size of a JVM stays constant during its lifetime. Dynamically throttling memory outside the JVMs is difficult. Inspired by [70], we adopt a method to enable dynamic memory limits at runtime by modifying OpenJDK [13]. As in operating systems, the virtual address space does not have physical memory until it is actually used, and the allocation module leverages this to reserve and commit specified address spaces of afi xed maximum heap size dynamically at runtime. We implement an API `JVMmanage()` inside a OpenJDK's JVM which listens to instructions from the allocation module for such dynamic memory commitment. As a result, each JVM knows the correct maximum memory size it can use at any time.

One may wonder that in case of prediction errors, out of memory (OOM) error [16, 70, 73] may happen due to insufficient memory. In fact, wefi nd that applications in Spark 2.1.0 do not just fail when memory is less than demanded since they can spill data to disk as a remedy. This of course slows down executors and would trigger reprovisioning, which then wouldfi x the problem.

Reprovisioning Module. This is implemented as a long-running process at the master node. It continuously collects reports about executor progress via monitor surrogates at each worker, and triggers reprovisioning by invoking the resource manager with the corresponding worker ID, application ID, executor ID, and correspongding resource time series. Subsequently, allocation module is instructed to adjust resource limits at runtime.

Prediction Module. Our current implementation simply uses the average resource time series of the latest 3 runs as the prediction result for recurring workloads with the same settings. Our analysis in §2.2 demonstrates it is fairly accurate. For applications with new settings Elasecutor has not seen, we rely on a prediction model based on SVR to infer the demand time series. Elasecutor can also leverage other prediction methods [17, 23, 26, 68, 75] which are beyond the scope of this paper. Note that for new applications which are never seen by the system before, we need to collect data for several runs before we can predict the resource demand time series. Until then, the applications are allowed to run with peak demanded resources.

SVR Prediction. We cast our prediction problem as a regression problem. We have $x_i, i = 1, 2, \ldots, n$, where x_i is the i-th multidimensional input vector that represents the application type, dataset size, and CPU and memory configurations, and n is the number of training samples. We also have the ground truth y_i which is the actual resource usage time series of the i-th run. As in §2.2, the executor's time-series resource usage is stable for the same application type and settings. The goal is to learn the relationship between x_i and y_i so that when a application with new settings submitted, we can predict its demand time series based on the model.

To do so, we rely on support vector regression (SVR) [25, 51, 63]. We use the radial basis function (RBF) kernel [42] which wefind to have better results than other kernels. We select ε-SVR [59] as the optimization model, andfi nd its optimal parameters ε based on k-fold crossvalidation and grid search [18] until the prediction reach the accuracy which we show in §5.5.

The prediction model is continuously trained online by successively collecting profiling results from RUD. Once an application with new settings is submitted, the model makes prediction about its resource usage time series and outputs the results for the scheduling module to consume. In our experiments, we found that our prediction process for applications with new settings can be done within 1s.

5 EVALUATION

We now evaluate Elasecutor using testbed experiments. Our evaluation answers the following questions:

- How much overall performance benefit can Elasecutor provide compared to existing solutions? (§5.2, §5.3)
- How efficiently does Elasecutor utilize resources? (§5.4)
- How well do the prediction, scheduling, and reprovisioning modules work? (§5.5)
- How much overhead does Elasecutor add? (§5.6)

5.1 Setup

Our testbed cluster consists of 35 machines connected with a 10 GbE switch. Each machine has two 2.4 GHz Intel Xeon E5-2630 v3 processors, 64 GB DDR4 RAM, a quad-port Intel X710 10 GbE NIC, and two 7200 RPM disks. All machines run Ubuntu 16.04.2 LTS with kernel version 4.4.0, Scala 2.10.4, and HDFS 2.6.0. We deploy our Elasecutor implementation on top of Spark 2.1.0.

Methodology. To test our prototype, we use eight workloads described in §2.2. The workloads are from the HiBench bigdata benchmarking suite [10], which are commonly used in existing work [33, 36, 54, 55]. For each workload we use different input data sizes listed in Table 3 and different CPU and memory upper limit configurations, ranging from one tofi ve cores for CPU and 2GB to 10GB for memory as in §2.2.

We generate 120 recurring applications with different workloads, input data sizes, and resource settings. In each experiment run, the same 120 applications are used. They arrive according to a Poisson process with mean inter-arrival time of 25 seconds for a period of 3200 seconds in each run. This is consistent with existing work [31, 33, 36, 54, 55]. Elasecutor's prediction module is fed with 3 runs of each application as explained in §4 and is used as the prediction results. Its SVR method has been trained offline with 30 runs of the recurring applications, and the same trained model is used for each experiment run. Besides the recurring applications, we prepare 12 applications with new input data size and new resource settings that has not seen by Elasecutor before. The same 12 applications are used in each run, but they arrive randomly within the period of 3200 seconds. Generally we repeat each experiment forfi ve runs unless stated otherwise.

Schemes Compared. We compare Elasecutor to three existing resource scheduling strategies which are deployed in production systems [31–33, 35, 67]. (1) ***Static***: This policy statically reserves CPU and memory for each executor according to the peak demand, and launches afi xed number of executors according to user request. (2) ***Dynamic***: This uses the built-in dynamic allocation policy in Spark to scale the number of executors dynamically based on the workload. Each executor is allocated a multiple of <1 core, 2GB RAM> [14]. This is similar to Morpheus [66] without preplanned time-varying reservations. We do not compare to Morpheus as it optimizes for SLO and load balance and is very different from Elasecutor. Both *static* and *dynamic* policies are implemented in Spark, and they only consider CPU and memory. (3) ***Tetris***: This policy considers network and disk I/O in addition to CPU and memory. Following [31], it greedily chooses an executor that has the highest dot product value between the vectors of machine available resources and executor peak demands, and allocates the peak resource demands to the executor. For all three schemes, we explicitly size related resources using `cgroups`.

Task Scheduling within an Application. For a given application, Elasecutor and all other schemes adopt the fair scheduler [9] for task scheduling. Other popular task schedulers [6, 19, 29, 31–33] can also be used by Elasecutor.

5.2 Makespan

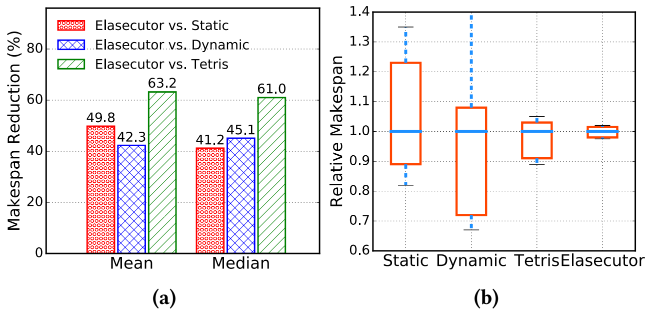


Figure 7: (a) Makespan reduction of Elasecutor; (b) Boxwhisker plot of makespans which are normalized to the median value of each scheme. Whiskers represent the maximum and minimum values. Each experiment run takes more than 5.9 hours, and we repeat 30 runs here.

Wefi rst investigate Elasecutor's makespan improvement.

We look at the makespan reduction provided by Elasecutor as shown in Figure 7a. Observe that Elasecutor reduces the average and median makespan by over 40% compared to existing schemes (60% over Tetris). Elastic resource allocation and scheduling in Elasecutor ensures that resources are not over-allocated or under-allocated during the executor's lifetime and machines are better utilized, thus translating to the smallest makespan performance. Also Elasecutor takes into account network I/O and disk I/O, which are not considered by *Static* and *Dynamic*. *Tetris* performs worse than *Static* and *Dynamic* because it has to wait until all four resources are available for the executor's peak demand, while *Static* and *Dynamic* only wait on CPU and memory. In fact, we observe in our experiments that applications run faster *after they start* in *Tetris* compared to *Static* and *Dynamic*, but they spend longer time waiting on resources.

Tetris's poor makespan performance is worth more discussion here. Actually its task assignment algorithm optimizes makespan, and is shown to have smaller makespan than DRF [29] and the capacity scheduler [6] in the paper [31]. The discrepancy of the results here is caused by two factors. First, Tetris along with DRF and capacity scheduler all use peak demand. Second, they are all designed for task-based systems like Hadoop, where tasks are short in duration and using peak demand isfi ne. When applying Tetris to executor-based systems in our systems, as executors last over the application's lifetime, Tetris has to wait until the peak demand can be satisfied for all four resources, and its performance degrades.

Figure 7b further shows the stability of makespan. Here we normalize the makespan to the median values under different policies over 30 runs. Recall that the same applications are submitted in each run with *random* arrival times as explained in §5.1. The result shows that Elasecutor delivers much more stable makespan than other policies with a much smaller box. In other words, Elasecutor is more consistent with respect to the arrival order of the applications. On the other hand, *Static* and *Dynamic* havefl uctuating makespan that depends heavily on the arrival order of the applications. This is because their peak-demand based allocation results in fragmentation that very much depends on application's arrival order, and their random executor assignment adds more inconsistency. *Tetris* has more consistent makespan by using a multi-resource packing heuristic to reduce fragmentation, but it still has similar problems due to the use of peak demand in the heuristic.

5.3 Application Completion Time

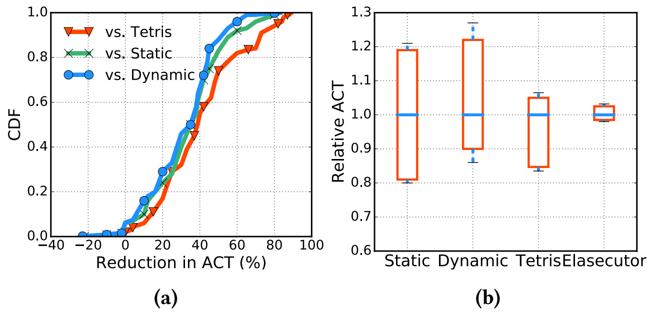


Figure 8: (a) The CDFs of reduction in application completion time of Elasecutor compared to *Static, Dynamic,* and *Tetris.* (b) The stability of average ACT under different policies. The average ACT across applications of the same run is normalized to median value under different policies.

We now look at the Elasecutor's improvement in application completion time (ACT), which represents user-perceived performance. Here ACT is defined as the time elapsed between application submission and completion. Figure 8a shows the CDFs of Elasecutor's ACT reduction over other policies. We can see that the median ACT reduction is 32.3%, 31.5%, and 39.6%, respectively, compared to *Static*, *Dynamic*, and *Tetris*. The top 20%ile applications are improved by over 49.1%, 45.3%, and 58.7%, respectively. The reduction over *Tetris* is larger compared to those over *Static* and *Dynamic*, again because *Tetris* waits on four resources for the executor's peak demand and increases ACT [31]. This also reflects that static peak allocation significantly hurts ACT performance.

We also investigate if Elasecutor can consistently provide better ACT. For each policy, we normalize an application's ACT in a run to the median ACT of this application across 30 runs, and plot the variability of such relative ACT across the 132 applications in Figure 8b. We observe that Elasecutor provides much more consistent ACT compared to other schemes. These results demonstrate that Elasecutor improves ACT and provides more consistent application level performance for users.

5.4 Resource Utilization

Now we investigate cluster resource utilization with Elasecutor. We show (estimated) resource usage with different policies in Figure 9

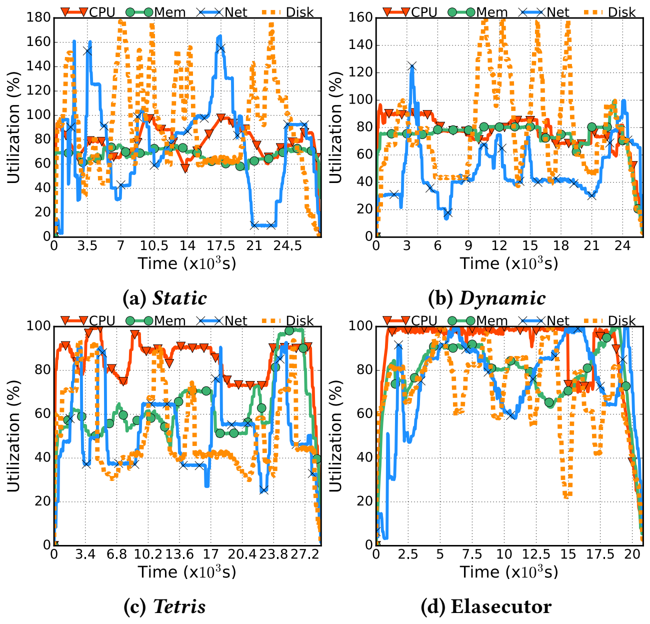


Figure 9: Resource utilizations under different policies. Note the time unit is 10^3s.

Policies	CPU (%)	Memory (%)	Network (%)	Disk I/O (%)
Static	61.3, 25.0, -	37.0, -, -	30.7, 25.1, 18.6	49.3, 37.2, 30.6
Dynamic	56.8, 26.6, -	57.7, -, -	25.3, 7.1, 5.0	51.2, 32.4, 26.9
Tetris	89.7, 70.8, -	49.1, 21.1, -	43.4, 13.3, -	37.2, 18.8, -
Elasecutor	97.0, 89.2, -	70.7, 39.4, -	68.7, 41.2, -	59.2, 37.0, -

Table 6: Frequencies that a machine's utilization in a resource exceeds a threshold. We use three thresholds 70%, 90%, and 100% here.

for one example run. For resources considered by a scheduling policy (CPU and memory for *Static* and *Dynamic*, all four for *Tetris* and Elasecutor), we measure and plot the actual utilization; for other resources, we derive the utilization based on our predicted demand time series. The same methodology is used in prior work [31, 33]. Observe that as in Figures 9a and 9b, *Static* and *Dynamic* are unable to fully utilize CPU and memory which they consider for executor allocation and placement. They also over-allocate network and disk resources most of the time, resulting in 180% utilization sometimes. *Tetris* in Figure 9c performs slightly better in that it avoids over-allocation. Elasecutor in Figure 9d improves the utilization of all resources with elastic resource scheduling. The cluster is bottlenecked on different resources at different times.

We also note that although *Dynamic* utilizes CPU and memory more, compared to Elasecutor it still loses 40% makespan performance as observed in §5.2. In addition to the more efficient *MinFrag* heuristic, the reason is that *Dynamic* incurs additional CPU and memory cost as it launches executors over time frequently. This also confirms our argument on the overhead of adjusting executor numbers over time in §1.

Utilization Improvement (%)	CPU	Memory	Network	Disk I/O
Elasecutor vs. *Static*	43.4	29.5	40.8	25.4
Elasecutor vs. *Dynamic*	27.2	22.6	33.4	40.0
Elasecutor vs. *Tetris*	28.6	25.2	55.6	43.9

Table 7: Elasecutor's average utilization improvement over other policies.

Table 6 depicts the fraction of time when a machine's utilization exceeds a threshold for each resource. We obtain the usage statistics from all machines with one example run. We can see that Elasecutor utilizes resources much more efficiently: for example 97% of the time CPU utilization is over 70%. On the other hand, *Static* and *Dynamic* sometimes over-allocate network and disk I/O, and waste CPU and memory that they statically reserved for executors.

Lastly, Table 7 shows the average utilization improvement with Elasecutor for all four types of resources. Compared to other policies, Elasecutor improves utilization by at least 27.2% for CPU, 22.6% for memory, 33.4% for network, and 25.4% for disk I/O. Clearly, this demonstrates Elasecutor can utilize resources efficiently and saves cost for cluster operators.

5.5 Microbenchmarks

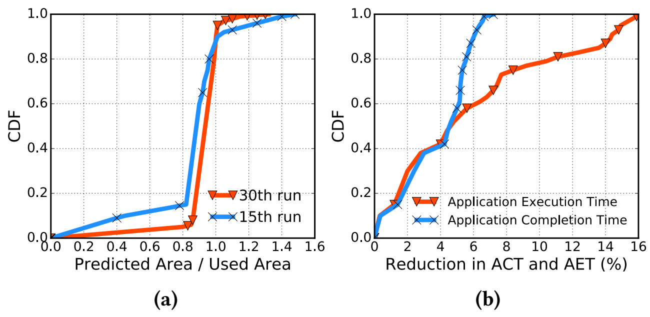


Figure 10: (a) CDFs of prediction effectiveness with more runs. (b) CDFs of reductions in application completion time (ACT) and application execution time (AET) by comparing Elasecutor with and without reprovisioning.

We now evaluate individual components of Elasecutor.

Effectiveness of Prediction. For the prediction module, we define prediction effectiveness as the ratio between the predicted total amount an executor is going to use and the actual total amount used during the execution for each resource. The average effectiveness across all 4 resources of each executor is then one data point. Note here we only concern executors of the 12 new applications with new settings. Figure 10a plots the CDFs of prediction effectiveness of the 15th and 30th run (with continuous training), respectively. We can see that the SVR method in §4 provides fairly accurate estimations: more than 80% of the time prediction is effective with less than 20% difference. Also with more training samples, the prediction improves with less than 15% errors more than 90% of the time (the curve for the 30th run).

Effectiveness of Reprovisioning. Now we estimate the effectiveness of the reprovisioning module. Figure 10b compares the CDF of the reductions in application completion time (ACT) and application execution time (AET) for Elasecutor with and without reprovisioning. Here AET is the time the application takes to complete after it is scheduled to run. The setup is the same as in §5.1 with 132 applications in each run for 5 runs. We can see that by using reprovisioning, Elasecutor's median ACT and AET decrease by 4.6% and 5.0%, respectively, and the 90%ile ACT and AET decreases by 6.0% and 14.5%, respectively, compared to not using it. The results demonstrate that reprovisioning is important for prediction based resource schedulers to improve application QoS performance.

Design Choices	Makespan			ACT		
	Average	Median	Stdev.	50th	90th	99th
DRR vs. TRC	11.5%	13.4%	5.7%	2.3%	6.1%	11.0%

Table 8: Improvement of DRR over TRC as an alternative metric for executor placement.

Effectiveness of DRR. Our scheduling heuristic *MinFrag* uses DRR. We now investigate its effectiveness by comparing with an alternative metric for multi-resource executor placement. Particularly, we consider to sum up the relative remaining capacity of each resource, i.e. $R(\text{CPU}, m)/C(\text{CPU}, m) + R(\text{Mem}, m)/C(\text{Mem}, m) + R(\text{Net}, m)/C(\text{Net}, m) + R(\text{Disk}, m)/C(\text{Disk}, m)$ for machine m, as the total remaining capacity (TRC). The scheduling module then applies TRC instead of DRR as the metric in the *MinFrag* heuristic in Algorithm 1. Effectively, in each iteration it chooses an executor that minimizes its TRC when placed on the current machine to run.

We compare DRR with TRC in Table 8 in terms of makespan and ACT. Observe that DRR reduces the average and median makespan by 11.5% and 13.4%, respectively, and have more stable makespan with 5.7% smaller standard deviation. DRR also improves ACT moderately in median and high percentiles. The results show that DRR works better than other alternative metrics to minimize resource fragmentation.

5.6 Overhead

Resource Consumption	CPU	Memory	Total executor profile size
Monitor surrogate	0.3%	0.1%	12.1 KB

Table 9: Resource consumption of a monitor surrogate. CPU and memory are averaged over all machines. Total executor profile size is the size of all executor profiles averaged across all machines at each heartbeat.

Time to process (ms)	Unmodified Spark	Elasecutor
Worker heartbeat	∼0.031	∼0.035
Application driver heartbeat	∼0.153	∼0.155

Table 10: Average time to process heartbeats from workers and application driver with and without Elasecutor over 100 heartbeats.

We now evaluate the overhead of Elasecutor. Wefi rst consider the overhead of monitor surrogate. Table 9 shows the average CPU and memory consumption of a monitor surrogate and the average size of executor profiles of all 35 machines of our cluster at each heartbeat. We can see that the overheads are very small, and the additional network overhead of sending profiles is negligible. We alsofi nd that the CPU and memory overhead of other modules of Elasecutor at the master node is less than that of the monitor surrogate.

We also evaluate Elasecutor's latency overhead to the analytics system, particularly the resource manager. The prediction, scheduling, and reprovisioning modules are independent processes running concurrently with the resource manager, and we wish to ensure they do not negatively impact resource manager's responsiveness. We measure the time taken by the resource manager to process a heartbeat from both a worker node and the application driver in Elasecutor and Spark. The application driver in Spark is responsible for creating context for executing applications, and we do not modify it in Elasecutor. Table 10 shows the results. Elasecutor takes about the same time as Spark to process both types of heartbeat messages.

6 DISCUSSIONS

After exhibiting the performance benefit of Elasecutor, we briefly discuss what we have done and the future work on Elasecutor in this section.

Model the resources jointly. In our current model, we do treat the executors' various resources (CPU, memory, network, and disk I/O) separately. Yet, in practice, they are often correlated. For instance, in Spark, the lack of available memory causes more spills and increases CPU (due to serialization overheads) and network or disk I/O (due to remote/local writes). Even though Elasecutor has dynamic reprovisioning mechanism (§3.3) to compensate for prediction errors caused by not considering such cases, we need to add such correlations between resources intro our model to make it work efficiently. Thus one future work for Elasecutor is to revise its resource demand prediction model and take correlations between resources into account.

Scalability. Due to the small scale of our testbed, the resource usage of the master node is very small. According to the average file size of an executor's profile on each server, the bandwidth used for transmitting them at each heartbeat is 32.27Kbps per server. Assuming 10Gbps network interfaces, the master node can support ∼309K machines. Besides, the CPU and memory costs are fairly low: both less than 1.5% to handle the current testbed scale. Correspondingly, if we fully utilize the CPU of a server, it can support 2.33K machines, which should be able to handle the typical production clusters [58, 62]. The scalability can be further improved in the following ways: (1) In production data centers 40G or 100G NICs are not uncommon [58, 62], which implies the bandwidth overhead of our system is even smaller compared to the 10Gbps links we use in our testbed; (2) We expect some nodes to have GPU, FPGA, or other hardware accelerators [41, 47] that can offload the computation from CPU and support larger clusters; (3) We can reduce the bandwidth requirement of updating executor resource profiles by adopting compression and/or sampling methods.

SLOs/SLAs. Elasecutor provides significant application completion time improvements and strives to provide better QoS via reprovisioning, but does not aim at guaranteeing *strict* SLOs/SLAs. To do the latter, Elasecutor would need to (1) have a dedicated component for SLO inference, (2) make resource reservations for executors ahead, and (3) dynamically adjusts resource allocations at runtime so that strict deadlines can be met [26, 66]. On the other hand, Elasecutor may be able to provide *statistical* SLO/SLA guarantees as it inherently predict applications' execution time based on historical data (§4). We are exploring this as future work.

Differences between DRR and DRF. DRR (dominant remaining resource) used in Elasecutor is inspired by DRF [29] and share some similarities, in particular the fact that they both use "dominant resource" to convert multi-dimensional metrics into scalars. Their differences, on the other hand, are distinct. DRR as defined in §3.2.2 concerns the maximum *remaining* resource of a machine *over time* and defines "dominant resource" based on it. DRF represents the maximum *time-invariant* resource *requirement* of a task with respect to the machine's capacity. Further, compared to DRF that aims to achieving fairness between tasks, DRR is used to reduce resource fragmentation and minimize makespan in *MinFrag*.

Other executor-based frameworks. Our current Elasecutor implementation is based on Spark. Streaming systems like Storm or Flink also have long running jobs, which potentially can also use Elasecutor. In the next step, we plan to explore other systems Elasecutor can be applied to, and make it an extension in resource managers like Yarn [67], Mesos [35], and Kubernetes [12] to serve all executor-based frameworks.

Workload usecase analysis. The ideal workloads for Elasecutor to obtain high performance are ones that either have a mix of I/O- and computing-intensive stages like Sort, Terasort, and Wordcount, or exhibit time-varying computation and I/O usage ratios like Pagerank, K-means, and Bayes. The executors used to run them can be packed together by Elasecutor to fully utilize various resources. In addition, deep learning training also often consists of long running tasks. When many training jobs co-exist in the same cluster, they progress in different paces and stress different resources (CPU, GPU, network, etc.), which makes it a goodfi t for Elasecutor as well. Another future work is then to run experiments for production deep learning training jobs to measure their actual resource usage patterns and investigate the potential benefit of Elasecutor there.

7 RELATED WORK

There has been a substantial body of work on scheduling and resource allocation in data analytics systems. We compare Elasecutor to related work along several dimensions as summarized in Table 11.

Granularity. Most of prior work [6, 9, 19, 24, 26, 29, 31–33, 38, 56, 78] assumes a task based system such as Hadoop. As discussed in §1 and §2, they do not work well for executors which run the application's entire DAG and have time-varying demands. Cluster schedulers such as Yarn [21, 67] and Mesos [35] manage the resource allocation for co-existing frameworks. They are also commonly used for task scheduling within frameworks in practice. Yet, they do not dig deeply to explore how to schedule the long-running executors within a computing framework and have the

Existing Work	Granularity	Scheduling			Objective
		Multiple Resources	Elastic Sizing	Machine Assignment	
Jockey [26], Quasar [24]	Task	CPU, memory	✗	✗	SLO
Sparrow [56], Apollo [19]	Task	CPU, memory	✗	✓	ACT and fairness
Tetris [31]	Task	✓	✗	✓	Makespan
Yarn [21, 67], Mesos [35], DRF [29], Omega [60]	Framework / Task	CPU, memory	✗	✗	Fairness
KOALA-F [46]	Framework	Servers	✗	✗	Utilization
Prophet [76]	Executor	Network, disk	✗	✓	Utilization
Morpheus [66]	Executor	CPU, memory	# executors	✓	SLO and load balance
Medea[27]	Executor	✓	✗	✓	Depend on objective functions
Elasecutor	Executor	✓	✓	✓	Makespan

Table 11: Summary of previous approaches compared to Elasecutor.

same problems as task schedulers when applied to executor scheduling. Prophet [76], Morpheus [66], and Medea [27] consider executor scheduling and are more related to our work.

Scheduling Considerations. Existing work can also differ in terms of their scheduling considerations. Most work focuses on CPU and memory only. Tetris [31], like Elasecutor, considers network and disk I/O in addition. Most task and cluster schedulers mentioned above adopt static allocation and do not elastically size the per-task resource allocation. Prophet [76] performs executor scheduling, but it only considers network and disk I/O and cannot support elastic resource sizing. Morpheus [66], another executor scheduler, dynamically adjusts the number of executors to meet application's demand, without changing the per-executor allocation though. This does not cope well with the multi-resource time-varying executor demand as we explained in §1 and brings large executor launching cost as we analyze in §5.4. Medea focuses on making good placement decisions but does not consider variations of resource utilization of long-running executors. As far as we know, Elasecutor is thefirst scheduler that elastically sizes executor resource allocation.

Lastly, for machine assignment, cluster schedulers [21, 29, 35, 60, 67] usually just use random assignment. Sparrow [56] uses randomized sampling to choose machines quickly. Other schedulers typically use packing heuristics but in different ways. Prophet [76] favors machines with the least sum of fragmentation and over-allocation. Morpheus [66] essentially uses the Worst Fit heuristic for SLO, and Tetris [31] uses the BFD heuristic based on the dot product of task's peak demand and machine's available resource capacity. Elasecutor adopts BFD with DRR inspired by DRF [29] and is shown to work more effectively than alternative metrics. Medea applies a mathematical optimization approach that accounts for constraints and global objectives. Clearly, Elasecutor's executor assignment problem can be formulated as Medea's constraints. A technical difficulty is that, as we showed in §3.2.1 our objective function and constraints are non-linear, while Medea employs linear programming to formulate its placement constraints.

Objective. In this regard, cluster schedulers [21, 29, 35, 60, 67] are the easiest to analyze: they optimize for fairness among co-existing frameworks. Task and executor schedulers optimize for various objectives: mostly makespan and ACT [56], and SLO or utilization. Elasecutor focuses on makespan and also improves ACT and utilization as experimentally shown in §5. Medea supports various objectives for long-running executors and it provides low placement latency for short-running executors.

Elastic VM Allocation. Finally we note that Elasecutor is also related to work such as PRESS [30], CloudScale [61], and AGILE [53] that predicts VM's resource usage time series online, and dynamically sizes resource allocation. However, they do not consider VM placement to proactively avoid resource conflict, and as a result, VMs can be left with insufficient resources at run-time and machines can be overloaded. As discussed in §3.2.1, multi-resource placement with time-varying demand is very challenging and requires careful heuristic design.

8 CONCLUSION

We have presented a novel executor scheduler Elasecutor. Elasecutor builds on the following two key ideas: elastically allocating resources to an executor to avoid over-allocation, and placing executors strategically to minimize multi-resource fragmentation. We prototype Elasecutor on Spark and evaluate it on a medium-scale testbed. Compared to existing approaches, Elasecutor reduces makespan by more than 42% on average and the median application completion time by up to 40%, while improving cluster resource utilization by up to 55%. We are working with a large internet company to test-deploy Elasecutor in one of their Spark clusters.

Going further, we are exploring some interesting directions. Placement constraints are common in practice [33, 45] and can be added to Elasecutor's scheduling heuristic. Elasecutor does not guarantee SLOs in terms of deadlines [26, 66] currently. We seek to better understand the impact of elastic multi-resource scheduling on application performance and SLO in order to support SLO guarantees. Finally, it is possible to develop a general executor scheduler based on Elasecutor so it can be integrated into Yarn and other cluster schedulers and benefit other executor-based systems.

9 ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd Patrick Stuedi for their valuable comments on this paper. This work was supported by the Research Grants Council, University Grants Committee of Hong Kong (award numbers C7036-15G and 11202315).

REFERENCES

[1] Apache Flink. http://flink.apache.org.
[2] Apache Hadoop. http://hadoop.apache.org.
[3] Apache Spark. https://spark.apache.org.
[4] Apache Storm. http://storm.apache.org.
[5] Apache Tez. http://tez.apache.org.
[6] Capacity Scheduler. http://bit.ly/1tGpbDN.

[7] Cluster Mode Overview. https://spark.apache.org/docs/2.1.0/cluster-overview.html.

[8] Elasecutor. https://github.com/NetX-lab/Elasecutor.

[9] Fair Scheduler. https://spark.apache.org/docs/2.1.0/job-scheduling.html#fair-scheduler-pools.

[10] HiBench. https://github.com/intel-hadoop/HiBench.

[11] How-to: Tune Your Apache Spark Jobs. http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2.

[12] Kubernetes. https://kubernetes.io/.

[13] OpenJDK. http://openjdk.java.net.

[14] Resource Allocation Policy in Spark 2.1.0. https://spark.apache.org/docs/2.1.0/job-scheduling.html#resource-allocation-policy.

[15] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing Data Parallel Computing. In *Proc. USENIX NSDI*, 2012.

[16] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote Memory in the Age of Fast Networks. In *Proc. ACM SoCC*, 2017.

[17] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Proc. USENIX NSDI*, 2017.

[18] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kegl. Algorithms for Hyper-Parameter Optimization. In *Proc. NIPS*, 2011.

[19] 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 *Proc. USENIX OSDI*, 2014.

[20] M. Chowdhury and I. Stoica. Efficient Coflow Scheduling Without Prior Knowledge. In *Proc. ACM SIGCOMM*, 2015.

[21] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based Scheduling: If You're Late Don't Blame Us! In *Proc. ACM SoCC*, 2014.

[22] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI*, 2004.

[23] C. Delimitrou and C. Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proc. ACM ASPLOS*, 2013.

[24] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proc. ACM ASPLOS*, 2014.

[25] H. Drucker, C. J. C. Burges, L. Kaufman, A. Smola, and V. Vapnik. Support Vector Regression Machines. In *Proc. NIPS*, 1996.

[26] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proc. ACM EuroSys*, 2012.

[27] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *Proc. ACM EuroSys*, 2018.

[28] M. R. Garey, R. L. Graham, and J. D. Ullman. Worst-case Analysis of Memory Allocation Algorithms. In *Proc. ACM STOC*, 1972.

[29] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proc. USENIX NSDI*, 2011.

[30] Z. Gong, X. Gu, and J. Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *Proc. IEEE CNSM*, 2010.

[31] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-Resource Packing for Cluster Schedulers. In *Proc. ACM SIGCOMM*, 2014.

[32] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic Scheduling in Multi-Resource Clusters. In *Proc. USENIX OSDI*, 2016.

[33] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *Proc. USENIX OSDI*, 2016.

[34] M. Grzegorz, H. A. Matthew, J. C. B. Aart, C. D. James, H. Ilan, L. Naty, and C. Grzegorz. Pregel: A System for Large-Scale Graph Processing. In *Proc. ACM SIGMOD*, 2010.

[35] 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 *Proc. USENIX NSDI*, 2011.

[36] C. Iorgulescu, F. Dinu, A. Raza, W. U. Hassan, and W. Zwaenepoel. Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling. In *Proc. USENIX ATC*, 2017.

[37] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proc. ACM EuroSys*, 2007.

[38] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proc. ACM SOSP*, 2009.

[39] E. G. Joseph, S. X. Reynold, D. Ankur, C. Daniel, J. F. Michael, and S. Ion. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proc. USENIX OSDI*, 2014.

[40] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proc. USENIX ATC*, 2015.

[41] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, A. Wated, and M. Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *Proc. USENIX OSDI*, 2014.

[42] R. Kondor and T. Jebara. A Kernel Between Sets of Vectors. In *Proc. ICML*, 2003.

[43] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Proc. OSDI*, 2010.

[44] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Proc. CIDR*, 2015.

[45] M. Korupolu, A. Singh, and B. Bamba. Coupled placement in modern data centers. In *Proc. IPTPS*, 2009.

[46] A. Kuzmanovska, R. H. Mak, and D. Epema. KOALA-F: A Resource Manager for Scheduling Frameworks in Clusters. In *Proc. IEEE/ACM CCGrid*, 2016.

[47] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proc. ACM SIGCOMM*, 2016.

[48] Y. Lu, A. Chowdhery, and S. Kandula. Optasia: A Relational Platform for Efficient Large-Scale Video Analytics. In *Proc. ACM SoCC*, 2016.

[49] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource Management with Deep Reinforcement Learning. In *Proc. ACM HotNets*, 2016.

[50] Z. Matei, D. Tathagata, L. Haoyuan, H. Timothy, S. Scott, and S. Ion. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proc. ACM SOSP*, 2013.

[51] C. Michele, R. Yan, and L. Zheng. Adaptive Kernel Approximation for Large-Scale Non-Linear SVM Prediction. In *Proc. ICML*, 2011.

[52] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *Proc. ACM SIGMOD*, 2010.

[53] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *Proc. USENIX ICAC*, 2013.

[54] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proc. ACM SOSP*, 2017.

[55] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making Sense of Performance in Data Analytics Frameworks. In *Proc. USENIX NSDI*, 2015.

[56] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proc. ACM SOSP*, 2013.

[57] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *Proc. ACM EuroSys*, 2016.

[58] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proc. ACM SIGCOMM*, 2015.

[59] B. SchOikopr, P. Bartlett, A. Smola, and R. Williamson. Shrinking the Thbe: A New Support Vector Regression Algorithm. In *Proc. NIPS*, 1999.

[60] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proc. EuroSys*, 2013.

[61] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems. In *Proc. ACM SoCC*, 2011.

[62] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proc. ACM SIGCOMM*, 2015.

[63] A. J. Smola and B. Schölkopf. A Tutorial on Support Vector Regression. http://www.svms.org/regression/SmSc98.pdf, Technical Report, 1998.

[64] J. Son, Y. Xiong, K. Tan, P. Wang, Z. Gan, and S. Moon. Protego: Cloud-Scale Multitenant IPsec Gateway. In *Proc. USENIX ATC*, 2017.

[65] 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 *Proc. ACM SIGMOD*, 2014.

[66] 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. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proc. USENIX OSDI*, 2016.

[67] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. ACM SoCC*, 2013.

[68] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *Proc. USENIX NSDI*, 2016.

[69] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale Cluster Management at Google with Borg. In *Proc. ACM EuroSys*, 2015.

[70] J. Wang and M. Balazinska. Elastic Memory Management for Cloud Data Analytics. In *Proc. USENIX ATC*, 2017.

[71] M. Weimer, Y. Chen, B.-G. Chun, T. Condie, C. Curino, C. Douglas, Y. Lee, T. Majestro, D. Malkhi, S. Matusevych, B. Myers, S. Narayanamurthy, R. Ramakrishnan, S. Rao, R. Sears, B. Sezgin, and J. Wang. REEF: Retainable Evaluator Execution Framework. In *Proc. ACM SIGMOD*, 2015.

[72] G. J. Woeginger. There is no asymptotic PTAS for two-dimensional vector packing. *Information Processing Letters*, 64:293–297, June 1997.

[73] W. Xia, H. Jiang, D. Feng, and Y. Hua. SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In *Proc. USENIX ATC*, 2011.
[74] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The Only Constant is Change: Incorporating Time-Varying Network Reservations in Data Centers. In *Proc. ACM SIGCOMM*, 2012.
[75] G. Xu and C.-Z. Xu. Prometheus: Online Estimation of Optimal Memory Demands for Workers in In-memory Distributed Computation. In *Proc. ACM SoCC poster*, 2017.
[76] G. Xu, C.-Z. Xu, and S. Jiang. Prophet: Scheduling Executors with Time-varying Resource Demands on Data-Parallel Computation Frameworks. In *Proc. USENIX ICAC*, 2016.
[77] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda. TR-Spark: Transient Computing for Big Data Analytics. In *Proc. ACM SoCC*, 2016.
[78] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proc. ACM EuroSys*, 2010.
[79] 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 *Proc. USENIX NSDI*, 2012.
[80] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. SLAQ: Quality-Driven Scheduling for Distributed Machine Learning. In *Proc. ACM SoCC*, 2017.