Improving Resource Utilization by Timely Fine-Grained Scheduling

Tatiana Jin
The Chinese University of Hong Kong
tjin@cse.cuhk.edu.hk

Zhenkun Cai
The Chinese University of Hong Kong
zkcai@cse.cuhk.edu.hk

Boyang Li
The Chinese University of Hong Kong
byli@cse.cuhk.edu.hk

Chenguang Zheng
The Chinese University of Hong Kong
cgzheng@cse.cuhk.edu.hk

Guanxian Jiang
The Chinese University of Hong Kong
gxjiang@cse.cuhk.edu.hk

James Cheng
The Chinese University of Hong Kong
jcheng@cse.cuhk.edu.hk

Abstract

Monotask is a unit of work that uses only a single type of resource (e.g., CPU, network, disk I/O). While monotask was primarily introduced as a means to reason about job performance, in this paper we show that this fine-grained, resource-oriented abstraction can be leveraged by job schedulers to maximize cluster resource utilization. Although recent cluster schedulers have significantly improved resource allocation, the utilization of the allocated resources is often not high due to inaccurate resource requests. In particular, we show that existing scheduling mechanisms are ineffective for handling jobs with dynamic resource usage, which exists in common workloads, and propose a resource negotiation mechanism between job schedulers and executors that makes use of monotasks. We design a new framework, called Ursa, which enables the scheduler to capture accurate resource demands dynamically from the execution runtime and to provide timely, fine-grained resource allocation based on monotasks. Ursa also enables high utilization of the allocated resources by the execution runtime. We show by experiments that Ursa is able to improve cluster resource utilization, which effectively translates to improved makespan and average JCT.

Keywords: Distributed computing; resource utilization

1 Introduction

Cluster resource utilization is an extensively studied topic [3, 9, 10, 13, 16, 21, 26, 34, 35, 37, 42, 43], but resource utilization can still be low for workloads with dynamic utilization patterns. In this paper, we make use of the notion of *monotask* [33] to design a new system, called *Ursa*, to improve resource utilization in the following two aspects: (1) *scheduling efficiency (SE)*, which measures how well a scheduler allocates resources to run jobs submitted to the cluster, and (2) *utilization efficiency (UE)*, which measures how fully the allocated resources are utilized by the running jobs.

Monotask is a unit of work that uses only a single type of resource (apart from memory), i.e., CPU, network, or disk. While monotask was originally introduced for job performance reasoning in MonoSpark [33], we show that it can be leveraged by job schedulers to maximize cluster resource utilization. In the following discussion, we first show the problems of existing systems in resource utilization, which motivate the adoption of monotasks in Ursa and Ursa's design. Then we highlight the challenges in achieving both high SE and UE.

Existing scheduler designs can be categorized as centralized [16, 21, 37, 42, 43] and distributed [3, 13, 34, 35], as well as a hybrid of them [9, 10, 26]. In centralized designs, a *centralized scheduler* maintains the resource utilization status of each server and allocates resources to jobs to achieve certain objectives (e.g., makespan [17, 18, 31, 51], fairness [6, 14, 24]). A job then launches containers (with allocated resources) on server nodes to run its tasks. In contrast to centralized coordination for resource allocation, *distributed schedulers* allow individual jobs to make independent scheduling decisions based on the cluster utilization status. Each server node maintains a task queue and tasks are placed in these queues according to the server load and resource requirements. The actual resource allocation and task execution are managed at each server locally. We discuss how different scheduler designs seek to improve cluster resource utilization below.

Low SE. The main causes to low SE include (a) *resource fragmentation* in bin-packing the resource demands of jobs according to servers' resource availability and (b) *scheduling latency* when making resource allocation decisions. To reduce resource fragmentation, effective algorithms [17–19, 31] that pack multi-dimensional resource demands (i.e., CPU cores, memory, network, disk) have been proposed for

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387551

centralized schedulers. However, since centralized schedulers allocate resources by communicating with servers via heartbeats, scheduling latency becomes a concern especially when short tasks dominate the workloads [3, 10, 26, 35, 49], which affects SE and hence cluster utilization. The "executor" model [28, 30, 32, 36, 44, 50] reuses containers that are allocated by a centralized scheduler to run other tasks, and so the scheduling cost is amortized across the tasks. However, the containers need to be large enough to accommodate the resource demands of all these tasks, which results in *coarser-grained resource allocation* and impairs UE when some tasks do not need to use all the resources in the containers. Compared with centralized scheduling, distributed schedulers hide the scheduling latency by allowing resources to be allocated by local servers without synchronization with each other. However, distributed scheduling lacks of global coordination in allocating resources to jobs, which may result in sub-optimal scheduling decisions.

Low UE. Even though recent schedulers have significantly improved SE, cluster utilization could still be low due to low UE, which is mainly caused by *resource under-utilization within a container* due to two factors: (a) *inaccurate container sizing* and (b) *highly dynamic resource utilization*. The problem of (a) is common as users tend to be conservative when estimating the peak resource demands, so that the allocated resources can be more than actually needed and consistently under-utilized during a container's lifespan. This problem can be alleviated by monitoring the actual utilization at runtime and launching preemptable opportunistic tasks to use the under-utilized resources [21, 26, 38, 48]. However, there is a non-trivial trade-off between (1) aggressively launching opportunistic tasks, which may result in a high preemption rate and hence wastes resources, and (2) a conservative strategy, which requires a longer period of resource under-utilization to be observed before launching an opportunistic task. There are also methods that predict resource usage by learning from historical workloads [8, 31], but it is costly to tune the model for heterogeneous workloads.

The problem of (b) commonly exists in in-memory machine learning, graph analytics, and OLAP workloads, in which a task first utilizes one type of resource (e.g., network to read remote data) and then shifts to use another type of resource (e.g., CPU to perform computation). The "executor" frameworks [28, 36, 50] can pipeline I/O and computation within tasks to overlap different types of resource usage. However, this creates many short periods in which resources are under-utilized as reported in [33]. In §2, we show that this type of resource under-utilization happens *at high frequency with short duration* (see Figure 1), which is not addressed by existing schedulers and can provide significant room for improving resource utilization.

Our solution. In order to achieve high UE, we propose a new mechanism between job schedulers and executors (§3), which makes use of the fine-grained, resource-oriented abstraction of monotask (in contrast to the coarser-grained resource allocation in existing schedulers) to communicate resource demands and allocate resources dynamically during job execution. On top of this mechanism for high UE, we achieve high SE by (1) addressing resource fragmentation by resource usage estimation based on monotasks (§4.2.1) and load-balanced task placement (§4.2.2), and (2) reducing scheduling latency by extending the scalable architecture of existing resource schedulers (§4.2.3).

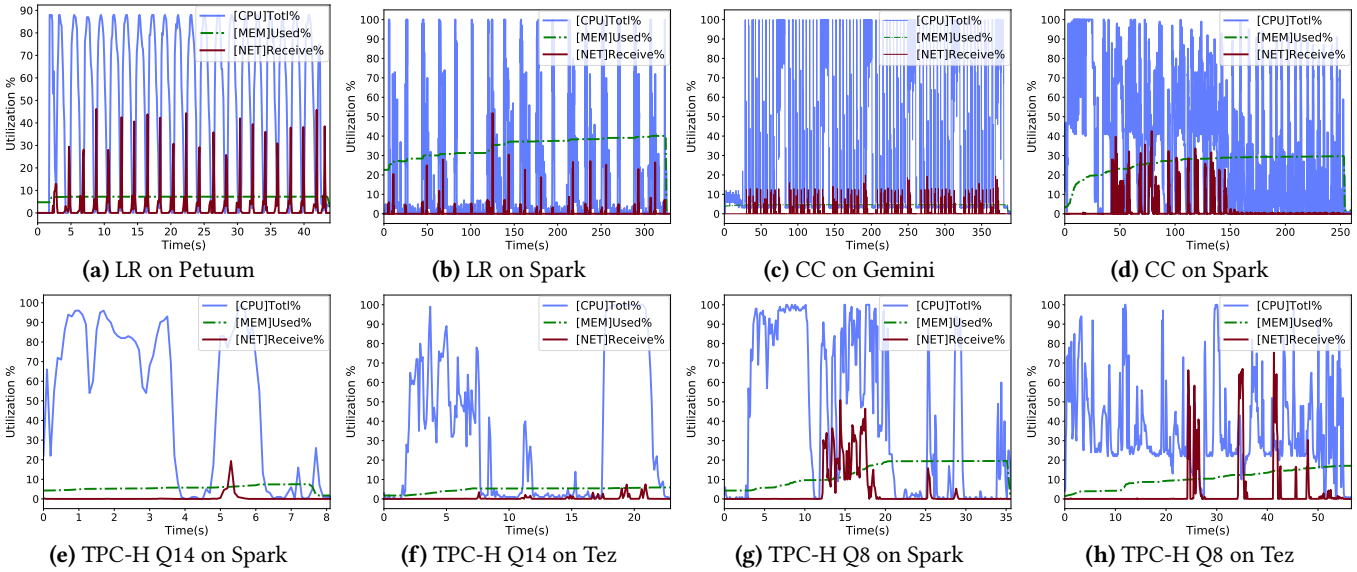
We implement our design in *Ursa*, a framework for scheduling and executing multiple, heterogeneous jobs [1]. For better programmability, Ursa provides high-level APIs such as Spark-like dataset transformations and SQL on top of its primitives for monotask specification (§4.1.2). To achieve high performance, Ursa provides low-latency, fine-grained resource allocation for executing monotasks (§4.2.3), as well as schedules jobs to overlap resource utilization so that each type of resource can be used as fully as possible (§4.2.2). Our experiments validate that Ursa's designs are effective in achieving high SE and UE, and consequently, significantly improved makespan and average JCT (§5).

2 Dynamic Resource Utilization Patterns

In this paper, we focus on the following types of workloads and analyze why it is challenging to achieve high UE for them: *OLAP*, *machine learning*, and *graph analytics*. The computation is in memory as fast response and high throughput are required. We ran TPC-H queries (with a scale factor of 200), logistic regression (LR), and connected components (CC) on 20 machines each with 32 cores and 128GB RAM connected by 10Gbps Ethernet. We executed the jobs using general dataflow systems, Spark [50] and Tez [36], with YARN [42] as the resource scheduler, and also using two domain-specific systems, Petuum [45] for machine learning and Gemini [52] for graph analytics. More details of the workloads and system configurations can be found in §5. Figure 1 reports their resource utilization patterns on one of the machines (all machines have similar patterns). Many other machine learning (e.g., k-means, ALS, SVM), graph analytics (e.g., PageRank, SSSP), and OLAP (e.g., TPC-DS) workloads also have similar patterns.

Dynamic patterns. First, Figures 1a-1d report one set of patterns: *regular and frequent alternation of very high and low CPU utilization*. Such patterns are due to the iterative nature of machine learning and graph algorithms, as in each iteration workers compute on data (mainly using CPU) and then communicate updates with each other (mainly using network). *For such workloads, even if the container size could be ideally tuned at exact peak resource demands (as we did in*

[1] Ursa supports popular data analytics workloads such as OLAP, machine learning and graph analytics.



(a) LR on Petuum **(b)** LR on Spark **(c)** CC on Gemini **(d)** CC on Spark

(e) TPC-H Q14 on Spark **(f)** TPC-H Q14 on Tez **(g)** TPC-H Q8 on Spark **(h)** TPC-H Q8 on Tez

Figure 1. Resource utilization of different workloads (best viewed in color)

this experiment), resources are still seriously under-utilized in alternate short time slices as shown in Figures 1a-1d.

Figures 1e-1h report another set of patterns: *irregular fluctuations in resource utilization.* Such patterns could be explained by the distribution of intermediate results at runtime, since a job may have tasks working on data with different skewness and thus have different resource demands (e.g., Q8 has many joins and group-by). *For such workloads, tuning container sizes for appropriate resource provision is difficult, thus leading to low UE.*

Low utilization. We also report the highest UE of CPU that can be achieved by Spark and Tez on YARN, by tuning the container sizes to the exact peak resource demands of various jobs. The CPU UE is calculated as follows:

$$\frac{Total\ CPU\ time\ used\ by\ the\ job}{Number\ of\ allocated\ CPU\ cores \times JCT}.$$

Table 1 reports the results. The low UE suggests much room to improve resource utilization. Maximizing CPU utilization is vital as CPU is the bottleneck for most common workloads, while both network bandwidth and memory in modern clusters are relatively more sufficient. However, when allocated CPU resources are unused in short time slices (sub-seconds to seconds) as shown in Figures 1a-1h, it is difficult to reallocate the under-utilized CPU to other jobs. On the other hand, naively overselling resources could result in serious load imbalance and resource contention (§5.1.2), which not only limits performance improvement, but also makes reasoning of job performance difficult.

To conclude, *frequent fluctuations in resource utilization pose significant challenges to existing container-based scheduling designs. To address this, resource allocation must be fine-grained and react to immediate resource demands at runtime.*

	LR	CC	TPC-H Q14	TPC-H Q8
Spark	13.97%	45.81%	62.16%	48.34%
Tez	N/A	N/A	30.93%	41.70%

Table 1. CPU utilization efficiency

Network contention. Network contention has been reported to be common in production clusters [3, 29]. Contention in network usage among tasks can block CPU utilization (§5.2) [2], as data shuffling and computation are always inter-dependent. Even if a cluster has sufficient network bandwidth (i.e., network resource is not a bottleneck for processing a workload), network contention can still happen locally at a worker when multiple jobs/tasks use the network stack simultaneously. Thus, coordination of data transfers among tasks both within and among jobs is required. The scheduler should learn the network resource demands of tasks from the execution framework at runtime, and consider the interplay between network utilization and CPU utilization to avoid CPU computation being blocked by network contention.

3 Design Objectives

Application scenario. Ursa is developed for the following application scenario. A company has many business units and each business unit has many projects. Each project is assigned quotas to use resources in a cluster. A project may have multiple *quota groups*, and applications in the same

[2]We remark that this is observed in processing workloads using commodity networking hardware such as 10GbE, but the case may change when modern hardware such as 40/100/200GbE is used. We discuss more details about the implication of modern hardware in §4.3.

group share resource quota. A *virtual cluster* is created for each group for security and resource sharing, which is allocated resources according to a quota (e.g., the computing power similar to a small cluster of tens of servers for some period) specified by a group. *Within a group, the most important objective is to minimize makespan (i.e., maximize job throughput) in order to save the overall project costs. Job completion time (JCT) is also important to users. But fairness is not the main concern of users in the same group as they share the same resource quota. Fairness among different groups, however, is important and guaranteed by a cluster scheduler (e.g., YARN) that allocates resources to all groups according to their quotas.* For example, the project groups described in [22] match the above scenario and their workloads are also mostly SQL OLAP workloads and some machine learning and graph analytics as the workloads in §2. We also found similar application scenarios and workloads in anti-money laundering projects with a financial services group and investment analysis projects with a multinational bank.

Design objectives. To address the issues identified in §2, we propose a fine-grained, dynamic resource negotiation mechanism using monotasks with the following objectives.

1. **Obj-1. Accurate resource request.** The execution framework should calculate each type of resource need (e.g., CPU, network, memory) based on monotasks at runtime, so as to request resources from the scheduler as accurately as needed.
2. **Obj-2. Timely provision and release of resource.** The execution framework should only request and obtain each type of resource when there are ready-to-run monotasks that use the resource, and each resource should be returned immediately when it is not used.
3. **Obj-3. Load-balanced task assignment.** The scheduling framework should schedule jobs and tasks in a way such that server nodes in the cluster have balanced load on each type of resource, while observing locality constraints acquired from the execution framework.
4. **Obj-4. Low-latency resource scheduling**. The latency, from resource release to allocation, should be low.

Obj-1 and Obj-2 together guarantee high UE. With high UE, we also need to have high SE in order to achieve overall high resource utilization. To achieve high SE, Obj-3 aims to keep all servers busy as long as there are unfinished tasks, while Obj-4 requires resources to be scheduled quickly when they are requested so that resources can be efficiently shared among jobs in a fine-grained manner.

Can existing methods achieve these objectives? Existing schedulers either pack resource requests of tasks using their peak demands (i.e., *task-based*) [3, 17–19], or pack containers where each container is an executor in an execution framework that runs multiple tasks (i.e., *executor-based*) [28, 31, 36, 44, 50]. Neither approaches can achieve

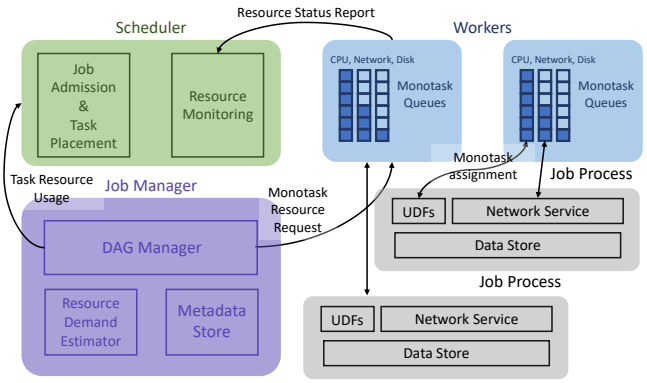


Figure 2. System architecture

Obj-1&2 when there is dynamic resource utilization within a container or a task. For example, when the CPU is used in alternate short intervals (e.g., patterns like Figures 1a-1d), the CPU resource is not (timely) released during the periods when CPU is idle, and the request for CPU is inaccurate because the request is made according to the peak demand of a task or tasks in a container. MonoSpark [33] enables timely per-resource scheduling through local per-resource queues, but this only optimizes the execution of tasks within a job, and MonoSpark still requests resources from a cluster scheduler as an executor-based solution.

For Obj-3, inaccurate resource requests naturally lead to imbalanced loads among worker nodes. Executor-based approaches [33, 44] may use runtime resource utilization information, but they do not have information of the future resource availability (e.g., resources to be released by other jobs) and can make sub-optimal task assignment decisions. In addition, task assignments among different jobs are not coordinated due to isolation.

4 System Design

Figure 2 shows the architecture of Ursa. Ursa integrates scheduling and execution runtime. Its scheduling layer consists of *a centralized scheduler* and *distributed queues*. The centralized scheduler handles job admission and task placement to workers. Each worker manages a queue for each type of monotasks and handles actual resource allocation. The execution layer consists of *job managers* (*JMs*) and *job processes* (*JPs*). Each job has one JM and multiple JPs, where the JM coordinates the execution flow of the job and communicates its resource requests to the scheduler and workers, and the JPs handle the execution of the job and update its execution status.

4.1 The Execution Layer

Ursa provides a simple set of primitives for specifying a dataflow. For better programmability, high-level APIs (e.g., Spark-like dataset transformations, SQL) are then built using

these primitives. Based on the primitives, a JM generates a physical execution DAG for a submitted job and coordinates the execution among it JPs.

4.1.1 Primitives. Each job is implemented as an operation graph *OpGraph*, which is formed of data *Dataset*, operations *Ops*, and dependencies among *Ops*. *OpGraph.CreateData*() creates a *Dataset*, which abstracts a distributed dataset with partitions. Using *OpGraph.CreateOp*(*type*), an *Op* that uses a single type of resource is created in *OpGraph*, where *type* is CPU, network, or disk. The parallelism of an *Op* is configurable based on the input data size, as in existing dataflow frameworks [36, 50]. An *Op* can *create*, *read*, and *update* a *Dataset*. A CPU *Op* contains a UDF to express data transformation (e.g., *map*, *filter*, *join*, *groupBy*). We use *Op*1.*To*(*Op*2) to create a dependency edge from *Op*1 to *Op*2 (e.g., *Op*2 consumes the output of *Op*1). We may specify the *type* of the dependency as either *sync* or *async*. The *sync* dependency imposes a synchronization barrier, such that *Op*2 can be executed only after *Op*1 finishes on all partitions of its *Dataset*. For *async*, *Op*2 can be executed asynchronously on any partition of *Dataset* as soon as *Op*1 finishes on that partition.

4.1.2 High-Level APIs. Ursa supports SQL with a plug-in to Hive [39]. Ursa also provides Spark-like dataset transformations including *map*, *flatMap*, *mapPartitions*, *groupByKey*, *reduceByKey*, *broadcast*, as well as a Pregel-like vertex-centric interface, so that users can specify common dataflows without dealing with the low-level primitives. We show an example implementation (in C++) of *reduceByKey* using our primitives as follows, while applications, e.g., the workloads tested in our experiments in §5, are implemented using our high-level APIs in a similar way as in existing dataflow systems such as Spark and Tez.

```
template <typename ValueType>
class Dataset { // ...
 auto ReduceByKey(Combiner combiner, int
     partitions) {
   auto msg = dag.CreateData(this->partitions);
   auto shuffled = dag.CreateData(partitions);
   auto result = dag.CreateData(partitions);
   auto ser = dag.CreateOp(CPU)  // create CPU Op
                 .Read(this).Create(msg)
                 .SetUDF(/*apply combiner locally
                     and serialize*/);
   auto shuffle = dag.CreateOp(Network)
                     .Read(msg).Create(shuffled);
   auto deser = dag.CreateOp(CPU)
                   .Read(shuffled).Create(result)
                   .SetUDF(/*deserialize and apply
                       combiner*/)
   this->creator.To(ser, ASYNC);
   ser.To(shuffle, SYNC);
   shuffle.To(deser, ASYNC);
   return result;
 }
```

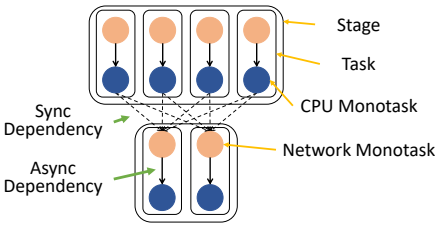


Figure 3. An example DAG

```
 // ...
 OpGraph dag;
 Op creator;
 int partitions;
};
```

4.1.3 DAG Management by JM. When a job is submitted, the scheduler creates a JM for the job by launching a process at a worker selected in a round-robin manner. The JM keeps the job's *OpGraph*, from which a DAG of monotasks is generated and maintained by the JM during task execution as follows.

Monotask generation. Each *Op* in *OpGraph* is transformed into a set of parallel monotasks that perform the same operation on different partitions of the same dataset(s). Depending on the *type* of *Op*, we have different types of monotasks: *CPU monotasks*, *network monotasks*, and *disk monotasks* as in [33]. As depicted in Figure 3, a *sync* dependency results in a many-to-many dependency between the monotasks of two *Ops*, which forms a fully connected bipartite graph. An *async* dependency forms a one-to-one dependency. For scalability in scheduling monotasks, the JM simplifies its DAG by collapsing a connected subgraph consisting of only CPU *Ops* (connected with *async* dependency) into one CPU *Op* before monotask generation.

Monotasks, tasks and stages. Given a DAG of monotasks, if we remove the in-edges of all network monotasks from the DAG, we obtain a set of connected components (CCs). As an illustration, if we remove the dashed edges in Figure 3, we obtain six CCs. As network transfer in Ursa is *pull*-based, *the monotasks in each CC should be collocated and executed at the same worker since they work on the data partitions that are to be pulled to the same worker.* Each CC is called a *task*, which is similar to a task in Spark and Tez. The set of tasks from the same *Ops* forms a *stage*.

Task execution flow. The JM maintains a list of *ready tasks*, i.e., *tasks with no parent tasks or whose parents have been completed.* For each ready task, the JM sends its estimated resource usage to the scheduler, and the scheduler assigns the task to a worker. The JM then sends the ready monotasks inside the task to the worker, and the worker puts the monotasks in the respective queues until resources become available for their execution. When a monotask is completed, the JM checks if the corresponding task is also

completed. If yes, the JM resolves task dependencies to update the list of ready tasks and continues the execution flow. Otherwise, the JM sends new ready monotasks inside the task to the same worker.

Metadata. The JM maintains a metadata store that records the size and locality of each dataset partition. When a monotask is completed, the information of the partitions it has created or updated is then updated in the metadata store.

4.1.4 Monotask Execution by JP. When a worker is first assigned to run the tasks of a job, it launches a JP for the job. The worker instructs the JP to execute the monotasks of the job when the requested resources become available. The JP reports to the JM and releases the resource to the worker when it completes a monotask. The JP maintains UDFs, network service, and a data store. When the JP receives a CPU monotask description, it launches a thread to run the UDF(s) on the input data partitions. When a network monotask is received, the JP sends a data transfer request to each sender JP that holds the remote data through the network service. Upon receiving the request, the network service of a sender JP starts to transfer the data. The JP invokes a disk I/O system call to execute a disk monotask.

4.2 The Scheduling Layer

The centralized scheduler (§4.2.2), JMs (§4.2.1), and workers (§4.2.3) all participate in the scheduling process.

4.2.1 Resource Request and Usage Estimation by JM. The JM of each job has two duties: (1) provide resource requests to the scheduler and workers for resource allocation (Obj-1&2), and (2) provide resource usage estimation to the scheduler for task placement (Obj-3).

Resource requests. Predicting the exact resource usage of a task (for making an accurate request) is difficult due to the highly dynamic resource utilization patterns of our target workloads as reported in §2. However, we can identify periods that have stable resource utilization during the execution of a task, and *hence make a separate resource request for each period.* JM acquires CPU, network and disk resources for a task on a *per-monotask* basis, because resource utilization within a monotask is more stable and predictable (e.g., no interleaving and I/O blocking). In particular, each CPU monotask uses exactly one core and has full utilization until it completes, i.e., the resource request is exact. JM sends the resource request for a monotask to its assigned worker only when the monotask is ready to run, so that the requested resource is immediately used once allocated. For memory requests, it is done on a *per-task* basis and simply uses the estimated memory usage (to be discussed below), since memory usage is relatively stable during the lifespan of a task as shown in Figures 1a-1h.

Resource usage estimation. The network and disk I/O usage of a monotask is estimated as the size of its input data. The CPU usage is calculated as the total amount of computation done by the monotask. As a monotask's CPU usage usually depends on the size of the input data, we simply estimate the CPU usage as the input data size as well [3]. The usage of a task is calculated as the sum of the estimated usage of all its monotasks. The input dataset partitions to be processed by CPU, or transferred through network (pull-based), or read from disk are known (and their sizes are recorded in the metadata store) at the time when the task becomes ready to run. Specifically, the sizes of the job inputs can be obtained from the filesystem (e.g., HDFS), and we use metadata in headers to calculate the uncompressed sizes in case the data are stored in compressed formats (e.g., ORC). Note that according to monotask generation in §4.1.3, there is at most one CPU monotask in each task and the input of the CPU monotask is simply read from network/disk by its parent monotask. For disk write, it is mostly for outputting the final result as we focus on in-memory workloads. For OLAP, machine learning and graph analytics workloads, the average output size of a job is much smaller than the input size (e.g., at least 29 times smaller for about 95% out of 7M jobs per day for such workloads in [22]). As disk is not a bottleneck, we simply estimate disk write usage as the total input size of the task. Currently, Ursa also does not consider cache for disk reads, which we leave for future improvement.

Existing schedulers [21, 26, 42, 43] usually rely on users to specify an estimated memory usage $M(j)$ for a job j. However, $M(j)$ is usually set much larger than the actual memory usage to avoid OOM or disk spilling. We introduce a configurable memory-to-input ratio $m2i$ to mitigate memory usage overestimation. Let $I(t)$ be the input size of a task t. The memory usage is then estimated as $min\{r \times M(j), m2i(t) \times I(t)\}$, where r is the ratio of the input size of t to the total input size of all the ready tasks of the job. The setting of $m2i$ is a trade-off between the memory utilization efficiency and the overhead incurred by disk spilling. Ursa provides an estimation of $m2i$ for recurring tasks based on their historical usage, in addition to a default setting of $m2i$ for some operations supported in Ursa's high-level APIs, e.g., $m2i = 2$ for *filter* and $m2i = 1 + s$ for *join* where s is the join selectivity obtained from the SQL query optimizer.

4.2.2 The Centralized Scheduler. The scheduler handles job admission and task placement.

Job admission. When a job is submitted, the scheduler creates its JM. The JM informs the scheduler of the estimated memory usage of the job. The scheduler admits the job if the cluster has sufficient memory, or otherwise puts the job in a queue. This is to prevent *memory deadlock*, as multiple jobs can be holding some memory while requesting more memory for task execution, but the remaining memory is insufficient

[3] Although different CPU tasks may have different complexity, we only use the input data size in JM and rely on processing rate monitoring by the scheduler to adjust for the difference (§4.2.2).

for any job to continue. Note that memory is not actually allocated from workers at job admission, but reserved cluster-wise to ensure that there is sufficient memory for a job to finish. When a job is admitted, its JM sends the estimated resource usage of its ready tasks to the scheduler for their placement to workers.

Task placement. The scheduler assigns tasks of jobs to workers. In order to achieve Obj-3, Ursa's task placement strategy differs from existing task scheduling algorithms [3, 17, 34, 35] in the following ways. First, we design a unified measure for multi-dimensional resource consumption and place tasks to maintain *per-resource load balancing* among workers. Second, to handle workloads with dynamic resource utilization, we use the *total resource consumption* in making placement decisions, in contrast to the *peak demands* of tasks used in existing work [3, 17]. Third, due to synchronization barriers in data shuffles between stages, even one straggler in a stage may block the execution of all tasks in the dependent stage(s). Stragglers arise when some tasks in a stage are not assigned to workers, thus their execution falling behind other tasks that are already placed to workers. Thus, we propose *stage-aware task placement* to avoid creating such stragglers. The details of task placement in Ursa are given as follows.

To measure the *per-resource load* of a worker w, we introduce $APT_r(w)$, i.e., the *approximate processing time* to complete all type-r monotasks currently assigned to w, where r is CPU, network or disk. We calculate $APT_r(w)$ as *the total amount of work of the monotasks divided by the processing rate of w using resource r*. The amount of work of a monotask is simply its input data size (§4.2.1). The worker periodically updates its *type-r processing rate* as X/T, where T is the accumulated execution time of type-r monotasks that were completed within the observed period and X is the total input sizes of the monotasks. If r is CPU, we multiply X/T by the number of CPU cores to indicate the overall CPU processing rate of the worker. In addition, if CPU in w is immediately available (i.e., idle cores), we simply set $APT_{cpu}(w) = 0$.

We also introduce *expected processing time*, EPT, which is used to quantify how overloaded or underloaded a worker is. The scheduler processes task placement in batches (for better placement and higher scheduling throughput) at a configurable scheduling interval. Thus, ideally EPT should be the same as the *scheduling latency* (i.e., the scheduling interval plus the delay in communication among the scheduler, JMs, and workers), such that within each interval of EPT, workers finish processing all assigned tasks and immediately start processing a new batch of tasks for the next interval. Thus, we set EPT slightly larger than the scheduling interval to account for the communication delay.

Then, we calculate the difference between EPT and $APT_r(w)$, normalized by EPT, as $D_r(w) = \max(0, \frac{EPT - APT_r(w)}{EPT})$. Intuitively, *a greater $D_r(w)$ means that an incoming type-r monotask needs to wait for less time to be executed at worker w and a small $D_r(w)$ indicates that resource r in w is being heavily used.* We also calculate $D_{mem}(w)$ as the size of available memory in w divided by the memory capacity of w.

Algorithm 1: TaskPlacement

```
1  foreach stage S whose tasks are to be placed do
2      (score_s, plan_s) ← StageScore(S, D)
3  Pick stage S with largest score_s, and place tasks in plan_s
4  function StageScore(S, D):
5      Initialize plan as an empty set of task-worker pair
6      Initialize score ← 0
7      Initialize stage_bonus to a large number
8      foreach task t in S do
9          Pick worker w_max ← arg max_w {F(t, w)}
10         if w_max does not exist then
11             stage_bonus ← 0
12         else
13             Add {t, w_max} to plan and update D_r(w)
14             score ← score + F(t, w)
15     return (score/|plan| + stage_bonus, plan)
```

Algorithm 1 describes how we use $D = \{D_{cpu}, D_{network}, D_{disk}, D_{mem}\}$ to assign tasks to workers. We compute a placement plan and a score for a stage of tasks each time, using the *StageScore* function. For each task t in the stage, let $Inc_r(t, w)$ be the increase in the load of worker w in using resource r if t is placed in w. For CPU, network and disk, $Inc_r(t, w)$ is given by the increase in $APT_r(w)$, which is calculated as t's estimated usage of resource r divided by w's type-r processing rate, normalized by EPT. $Inc_{mem}(t, w)$ is simply the estimated memory usage of t.

The scheduler decides whether a task t should be placed in a worker w by measuring how similar Inc and D are. If w has sufficient memory to run t, a score $F(t, w)$ is calculated:

$$F(t, w) = \sum_{r \in \{cpu, network, disk, mem\}} D_r(w) \times Inc_r(t, w). \quad (1)$$

We match t with the worker w that gives the highest $F(t, w)$. According to Eq. (1), a larger $Inc_r(t, w)$ (i.e., a heavier load of resource r) and a larger $D_r(w)$ (i.e., a lighter load of r at w) contribute more to $F(t, w)$. However, it is possible that w is overloaded with one resource but not with the other. For example, $D_{cpu}(w) = 0$ but $D_r(w)$ is large for the other resources, which may still give a large $F(t, w)$. In this case, assigning t to w would result in its execution being blocked by waiting for CPU. To avoid this, we do not assign t to w if $D_r(w) = 0$ and $Inc_r(t, w) > 0$ for any r. It is also possible that $Inc_r(t, w) > D_r(w)$, in which case we still consider w but reset $Inc_r(t, w) = D_r(w)$ to indicate that the resource availability is bounded by $D_r(w)$, which also bounds the contribution of $D_r(w) \times Inc_r(t, w)$ to $F(t, w)$.

To enforce stage-awareness, Algorithm 1 gives a large bonus to $plan_s$ if the plan assigns *all* tasks in stage S, so that such plans are always considered before other plans.

Job ordering. The design of Algorithm 1 is more for maximizing resource utilization and hence job throughput, but does not consider average JCT. To reduce average JCT, Ursa supports two scheduling policies to order jobs: *Earliest Job First (EJF)* and *Smallest Remaining Job First (SRJF)*.

The scheduler performs job admission by first considering jobs that are submitted earlier. Starvation of jobs that have large memory requirement is handled similarly as in existing schedulers [35, 42]. In task placement, Ursa prioritizes earlier jobs by the elapsed time T since the submission of a job, and adds WT to the placement score of each stage of the job, where W is a weight that indicates how much EJF should be enforced.

For recurrent workloads where historical resource usage information is available, JCT can also be reduced by prioritizing jobs with smallest remaining work, based on a similar concept in [35]. However, instead of estimating the total time needed to complete the remaining tasks of a job, Ursa uses *the total remaining per-resource work of a job*, R, and *the per-resource cluster load*, L, to rank the jobs. R and L are vectors, where $R[r]$ and $L[r]$ are for resource type r. R is initialized based on historical information and is updated by the JM whenever a monotask of the job is completed. L is calculated as the total remaining work of all admitted jobs. The priority score of a job for applying SRJF is then calculated as the inverse of the dot product of $(2L - R)$ and R normalized by L. The intuition is that when a resource r is heavily demanded, more weight is given to r to pick the job with the smallest remaining work. The enforcement of SRJF in job admission and task placement is similar as in EJF.

4.2.3 Distributed Queue Management.

Distributed queues are used to minimize the latency of allocating a resource to a monotask when the demanded resource becomes available at a worker. Each worker maintains a queue of monotasks for each resource (i.e., CPU, network, and disk) and decides the order and concurrency of resource allocation to monotasks.

Monotask ordering. Instead of FIFO, monotasks in each queue are ordered based on the scheduling policy and task dependency. Among jobs, monotasks are ordered according to their job priorities (EJF or SRJF). Within a job, CPU monotasks in the same stage are ordered in descending order of their input sizes so that larger tasks can start earlier to reduce the completion time of the stage, while network and disk monotasks in the same stage are ordered in ascending order of their input sizes to make their dependent monotasks ready earlier.

Concurrency control. While the number of CPU monotasks (possibly of different jobs) being executed concurrently is simply set as the number of CPU cores, the concurrency of network and disk monotasks running at a worker is a trade-off between maximizing resource utilization and avoiding contention. The concurrency limit for disk monotasks is set as one monotask per disk, which can already fully use the I/O bandwidth when transferring a considerable amount of data (sequentially). In cases when there are many small disk monotasks, we run them together and reply on the OS to optimize the seek time (by I/O request re-ordering).

The concurrency control for network monotasks is challenging as the bandwidth usage of a network monotask depends on the amount of data flowing through both the sender and receiver machines. It is too costly to model the all-pair data flows and calculate the concurrency that maximizes network utilization, especially at the fine granularity of monotasks. We use a simple method that considers only the network bandwidth at the receiver side. Each network monotask pulls data from all its senders at the same time, so that the receiving bandwidth can be more fully utilized even if congestion may occur at some senders. Although data transfers from different senders may contend at the receiver downlink, this does not add additional delay. To avoid contention among network monotasks, a small concurrency of 1 to 4 is set for each worker. In addition, Ursa allows latency-sensitive small monotasks (typically smaller than 16KB) to be executed without being queued.

4.3 Discussion

Working in multi-tenant clusters. While Ursa is designed to enable fine-grained resource sharing for jobs in a virtual cluster, it does not require static partitioning in a large cluster. Ursa can work with YARN as an execution framework through YARN's APIs for ApplicationMaster (AM). The scheduler of Ursa runs on the AM container and communicates with the AM for resource negotiation with YARN, while Ursa launches workers and jobs in the allocated containers. The workers are informed by the scheduler of the allocated containers on the local nodes. We remark, however, as YARN does not have native management for network resources, Ursa does not consider network usage of concurrent applications on YARN but relies on processing rate monitoring by its scheduler to adjust network workloads among workers (i.e., through APT).

Implications of modern I/O hardware on Ursa's design. As Ursa's designs are mainly motivated by the dynamic resource utilization patterns discussed in §2, it is critical to consider the possible influence of modern I/O hardware on the resource utilization patterns and how it may influence Ursa's designs. We consider this problem from three perspectives: load balancing of resource usage, use of monotasks, and types of resources.

First, Ursa's scheduling layer is designed with the use of monotasks to avoid blocking the utilization of other types of resources when task execution is (temporarily) bottlenecked on one type of resource, e.g., the problem found in §2 is mainly to avoid blocking the CPU utilization by local network contention or stragglers in a stage. One key challenge is to maintain load balancing of different types of resources

across workers while observing data dependency of monotasks. Thus, we expect that the resource usage estimation and load-balanced task placement of Ursa using monotasks' resource-oriented work abstraction and runtime information of (intermediate) data will remain applicable even when modern I/O hardware is used, although the detailed methodology of resource usage estimation may require some adaption to possibly new resource utilization patterns.

Second, while monotask assumes the usage of only one type of resource, this assumption may require more consideration with the use of modern I/O hardware. For example, while RDMA enables bypassing CPU in sending messages, a network monotask may incur high CPU utilization when pulling messages over a 40/100/200 GbE link without RDMA. However, we emphasize that Ursa's design does not rely on the assumption of full utilization of a single type of resource by monotasks, but that a "monotask", or simply a task, should have stable resource utilization during its execution as discussed in 4.2.1. Therefore, the notion of monotask can be extended in case of new I/O hardware to a unit of work that utilizes one or multiple types of resources stably during a period of execution. Similarly, the Op in Ursa's primitives can be extended to support different types of "monotasks" with respect to the hardware to be used.

Third, while an operation involves many parts of the operating system, the abstraction of resource usage is often simplified by recognizing the bottleneck resource in the system stack. Particularly, sending a message through the network not only involves using network bandwidth, but also using CPU and memory bandwidth to copy data through the kernel networking stack. As network bandwidth has been the bottleneck of sending messages for many existing hardware, it is often used by resource managers to quantify the resource usage on the networking stack. However, with modern 40/100/200 GbE network, memory bandwidth might become the bottleneck instead. Therefore, the types of resources considered in scheduling should also be extended to account for new bottlenecks in processing workloads using modern hardware.

Fault tolerance. Ursa adopts a simple mechanism using heartbeats for detection and checkpoint for job recovery as in existing systems [42, 50]. Other techniques such as hot standby and lineage-based recovery may also be used.

5 Performance Analysis

The experiments were run on a cluster with 20 machines connected via 10 Gbps Ethernet. Each machine has two 2.1GHz Intel(R) Xeon(R) Silver 4110 CPU with hyper-threading enabled (32 virtual cores), 128GB RAM, and a 1.2TB SAS disk (6Gb/s, 10k rpm, 128MB cache), running on 64-bit Debian release 9.8. Note that Ursa is not a cluster scheduler that manages thousands of machines, but usually runs on a small (virtual) cluster requested by a quota group (§3).

Performance metrics. We mainly report makespan and JCT. We also measure average scheduling efficiency (SE) and average utilization efficiency (UE) for CPU and memory. Let X be *the allocated core/memory time*, Y be *the total core/memory time (i.e., total cores/memory multiplied by makespan)*, and Z be *the actual resource utilization time*. SE is defined as X/Y and UE is Z/X. The average resource utilization rate of the cluster is equivalent to (SE $\times$ UE).

Workloads. The **TPC-H** workload consists of 200 jobs and 3 datasets of size 200GB, 500GB and 1TB. The 200 jobs are 200 queries chosen uniformly at random from the set of TPC-H queries [40], and each job is run on the 200GB, 500GB and 1TB datasets with a probability of 60%, 30% and 10%, respectively. The depth of the DAGs of these jobs ranges from 2 to 10. When executed individually, the JCT of these jobs ranges from 3 sec to 297 sec (mean: 37.78 sec) and that of their tasks ranges from a few msec to 130 sec (mean & 80th percentile: 5 sec). The **TPC-DS** workload consists of 200 jobs and 3 datasets of size 200GB, 500GB and 1TB, and is created similarly as TPC-H. The depth of the DAGs ranges from 5 to 43 (mean: 9). When executed individually, the JCT of these jobs ranges from 9 sec to 212 sec (mean: 48.23 sec). These numbers are quite representative of our target production workloads, i.e., many heterogeneous and short tasks.

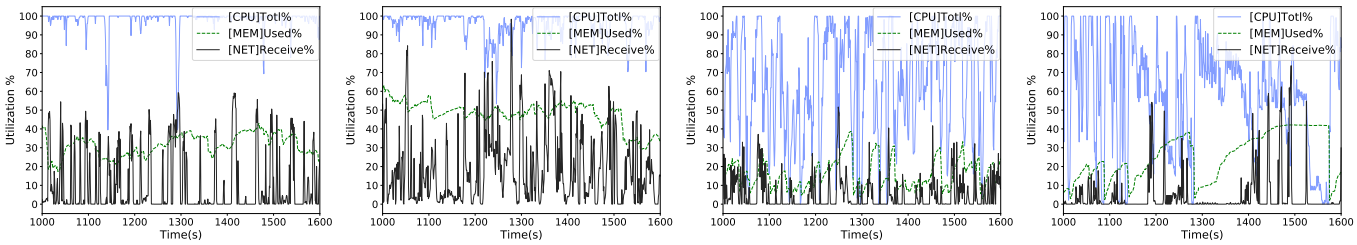
We also created a mixed workload, **Mixed**, which consists of 2 graph analytics (PR on WebUK [1] and CC on Friendster [47]), 4 machine learning (k-means on mnist8m [2] and LR on webspam [5]), and 32 randomly chosen TPC-H queries. The jobs are chosen such that TPC-H, ML and graph jobs account for 70%, 20% and 10% of the total CPU usage, which simulates our target production workloads.

We used a smaller TPC-H workload, **TPC-H2**, which consists 25 jobs with relatively deeper DAGs (average depth: 7.2) and more heterogeneous tasks with irregular utilization patterns. It is more challenging to achieve high resource utilization for these jobs and we used them to assess the effectiveness of various techniques used in Ursa.

5.1 Resource Utilization Analysis

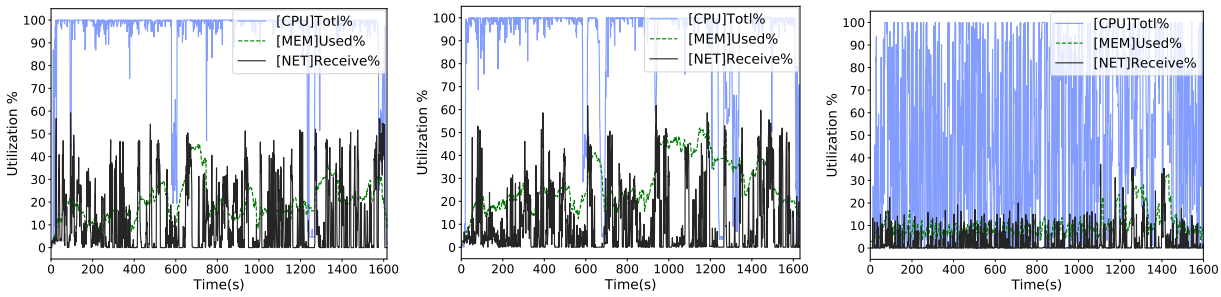
Since Ursa is an integration of scheduling and execution frameworks, we compared with **(YARN+Spark)** and **(YARN+Tez)**, denoted by **Y+S** and **Y+T**, where YARN is used as the resource scheduler and Spark/Tez as the execution framework. We also simulated MonoSpark [33] in §5.1.2. We do not compare with Spark and Tez alone because they are not designed to schedule multiple jobs for high resource utilization in a cluster. *Our goal is to demonstrate the limitations of using coarse-grained containers as resource scheduling units and examine the effectiveness of Ursa's design choices.*

5.1.1 Performance on TPC-H and TPC-DS. We first used the TPC-H workload, by submitting a new job at every 5 seconds to simulate a heavy online workload in our application scenario. We used YARN 2.8.5 with Spark 2.4.0



(a) Ursa-EJF **(b)** Ursa-SRJF **(c)** Y+S **(d)** Y+T

Figure 4. Resource utilization for TPC-H



(a) Ursa-EJF **(b)** Ursa-SRJF **(c)** Y+S

Figure 5. Resource utilization for TPC-DS

and Tez 0.9.1. We enabled the FIFO job scheduling policy in YARN, and set the heartbeat interval to 1 second. We tested both EJF and SRJF in Ursa. Note that FIFO in YARN and EJF in Ursa are essentially the same policy as both of them prioritize earliest jobs, except that EJF in Ursa is enforced in a more fine-grained manner. In contrast, SRJF is a different policy and we used it to demonstrate how Ursa's fine-grained scheduling can be adapted to serve a different policy to improve average JCT. The number of CPU cores per container was carefully tuned for Spark and Tez. For the container memory size, we ensured that memory is sufficient for all jobs. For Spark, we configured the container size to be 4 cores and 8 GB memory, and enabled dynamic allocation with idle timeout of 2 seconds. For Tez, we configured each container to have 2 cores and 6 GB memory, and enabled container reusing across tasks. Many other configurations were also tested and the above one gave the best overall performance for both Spark and Tez. For example, while using smaller containers for Spark leads to better resource utilization, it impairs job performance and results in longer makespan. We also tested Ursa and Y+S on the TPC-DS workload using a similar setting, except that the dynamic allocation idle timeout for Spark was set to 5 seconds for better performance.

Results on TPC-H. Table 2 reports the results for Ursa using EJF and SRJF, Y+S and Y+T on the TPC-H workload. While both Ursa and YARN achieve high CPU SE (memory SE is not high as memory is rather sufficient), Ursa achieves significantly higher UE (for both CPU and memory) than both Y+S and Y+T. We believe the higher CPU utilization is

	makespan	avgJCT	UE_{cpu}	SE_{cpu}	UE_{mem}	SE_{mem}
EJF	2803	600.00	99.64	92.47	78.83	39.80
SRJF	2859	489.96	99.65	89.73	78.02	48.85
Y+S	3849	1407.40	69.35	93.32	34.69	44.13
Y+T	9228	4287.00	58.97	98.19	28.81	70.71

Table 2. Performance on TPC-H (Col 2-3: sec; Col 4-7: %)

	makespan	avgJCT	UE_{cpu}	SE_{cpu}	UE_{mem}	SE_{mem}
EJF	1613	453.20	99.57	88.31	81.64	25.01
SRJF	1630	242.27	99.75	86.99	85.83	32.93
Y+S	2927	894.36	48.56	90.48	19.39	37.65

Table 3. Performance on TPC-DS (Col 2-3: sec; Col 4-7: %)

the main reason that leads to the shorter makespan compared with Y+S, because the total CPU time consumed by Spark and Ursa are actually comparable. Note that if the total CPU consumption (core $*$ time) of processing a workload is fixed, then makespan and CPU utilization are reversely correlated, i.e., to achieve a shorter makespan, a higher ($UE_{cpu} * SE_{cpu}$) is necessary.

Figures 4a-4d further reports the resource utilization rates[4] of the systems for processing TPC-H. The results show that Ursa achieves rather consistent high CPU utilization (the utilization on network and memory is not high because they are both sufficient). In contrast, the CPU utilization of the

[4]We only plot a 10-min interval for better visual presentation, and similar patterns are also observed for other time intervals.

executor-based solutions, Y+S and Y+T, fluctuates significantly and frequently.

The problem with the *executor-based* solutions compared to Ursa is that the resources allocated to a container are not immediately released when unused, but only released when there is no task to run in the container. Thus, containers are under-utilized when the running tasks cannot use up the allocated capacity and the unused resources cannot be immediately reallocated to other jobs. We tried to mitigate the problem by reducing the container size in Spark or disabling container reuse in Tez (i.e., resorting to *task-based* resource requests). This improved the UE, but resulted in even worse makespan and average JCT due to less process-level optimization (e.g., process-level cache for broadcast join) [7]. Ursa avoids the above problems by accurate resource requests (§4.2.1), timely provision and release (§4.2.3, §4.1.4) of different types of resources for each process in the granularity of monotasks.

Results on TPC-DS. Figure 5 and Table 3 reports the results for Ursa (using EJF and SRJF) and Y+S on TPC-DS. The resource utilization of Ursa on TPC-DS is similar to that on TPC-H, while the CPU utilization of Y+S on TPC-DS is considerably lower than that on TPC-H. The makespan and average JCT of Y+S are significantly larger than those of Ursa . The difference in performance is due to partitioned tables and much deeper DAGs in the TPC-DS workloads. When partitioned tables are involved, Spark jobs in TPC-DS incur extra costs in listing the leaf files and directories, which impairs JCT. Also, for small datasets (e.g., 200GB), some partitioned tables have many small partitions, which lead to greater overheads because of scheduling and executing many small tasks on Y+S. For deep DAGs, Spark jobs have many stages with alternating high and low parallelism (e.g., in one job it goes from 3,367 tasks to 1,090 tasks, and then to 2,791 tasks). Even though we enabled dynamic allocation in Spark, the idle containers may not be released in the stages with small parallelism as these stages are shorter than the timeout, which leads to bad UE. Meanwhile, in the case when the idle containers are released in the small stages, the Spark job needs to request containers when it progresses to a large stage, which not only leads to scheduling delay but also makes Spark to schedule tasks to existing containers with inferior locality and thus incur I/O overhead.

5.1.2 Performance on the Mixed Workload.

We want to examine *(1) whether the use of monotasks, or the execution framework, is the main contributor to Ursa's performance, (2) whether an existing scheduling algorithm can achieve better results, (3) whether over-subscription of CPU can achieve high resource utilization.* We used the more challenging Mixed workload that contain ML, graph and SQL jobs for this set of experiments.

Is monotask sufficient? We ran **Ursa on YARN** (**Y+U**) following the executor-based solution to validate that *Ursa's performance advantage mainly comes from better resource sharing among jobs instead of the use of monotasks alone or the execution layer implementation.* We implemented an application master for Ursa similar to that of Spark, so that each job requests resources from YARN instead of from Ursa's own scheduler. We launched one instance of Y+U for each job using the same container sizes as the corresponding Spark job. This single-job instance of Ursa simulates **MonoSpark** [33] as we enabled local per-resource queues except that we added monotask ordering.

	makespan	avg JCT	UE_{cpu}	SE_{cpu}
Ursa-EJF	464.00	208.21	99.57	86.60
Ursa-SRJF	473.50	170.64	98.89	86.08
Y+U	842.92	443.80	44.15	89.97
Y+S	1072.66	435.00	67.92	83.84
Capacity	511.00	226.16	99.77	78.66
Tetris	562.33	254.52	98.62	70.02
Tetris2	506.00	240.83	99.71	79.75

Table 4. Performance on Mixed (Col 2-3: sec; Col 4-5: %)

Table 4 shows that although Ursa and Y+U share the same execution layer that uses monotasks, Y+U has poor UE as it suffers the same problems of executor-based solutions (§5.1.1). This shows that fine-grained resource sharing among tasks within a single job is not sufficient, but fine-grained resource sharing among jobs is also needed in order to achieve high resource utilization. Y+U and Y+S have comparable makespan and average JCT, but both are significantly worse than Ursa. Thus, the better performance over Y+S is not because Ursa has a better execution layer than Spark, but because Ursa enables timely fine-grained resource allocation, which leads to high UE.

Alternative scheduling algorithms. We replaced Algorithm 1 in Ursa with Tetris [17], a state-of-the-art scheduling algorithm that aims to minimize resource fragmentation by packing multi-dimensional resource requests, and YARN's Capacity scheduling algorithm [20], which greedily assigns tasks to workers that have more available resources. For Tetris, we collected the peak demands of tasks in previous runs as discussed in [17] and used them for resource requests. We also tested Tetris by ignoring the network demand, denoted by Tetris2.

Table 4 shows that Ursa using Tetris and Capacity achieve good performance, though not as good as Ursa itself (i.e., using Algorithm 1). The performance difference is reflected in their SE_{cpu}, which means that the task placement algorithm affects resource allocation on workers. One key difference is that Tetris and Capacity use peak resource demands, while Algorithm 1 uses the estimated total resource usage. When a monotask of a task completes, the worker updates the per-resource load to the scheduler so that Algorithm 1

subscription ratio	makespan (Y+U)	avg JCT (Y+U)	makespan (Y+S)	avg JCT (Y+S)
1	842.92	443.80	1072.66	435.00
2	637.96	345.99	872.67	341.77
4	596.66	325.32	892.83	365.30

Table 5. Results of CPU over-subscription (sec)

can assign new tasks to use the released resource. However, Tetris and Capacity do not make use of such information but let the worker wait until the entire task finishes. Thus, using peak resource demands is not suitable for measuring the resource usage of jobs with frequent fluctuations in resource utilization. This is further verified by the result that Tetris2 actually outperforms Tetris, as we found that task assignment is blocked when a task's peak network demand exceeds the available network bandwidth, even though the network is not being used most of the time. This prevents workers to allocate available CPU resources to new monotasks, which results in 9.73% lower SE_{cpu} than Tetris2.

Over-subscription of CPU. We further show that simply oversubscribing CPU does not solve the problem caused by dynamic utilization patterns (§2), but only enables resource overlapping to a limited degree. We ran both Y+U and Y+S with CPU subscription ratio of 1 (no over-subscription), 2, and 4. We set the container size to 4 GB memory for SQL jobs so that we have enough memory to assign up to 4 times more containers for over-subscription.

Table 5 shows that both makespan and JCT improve most with an over-subscription ratio of 2, which is a direct consequence of improved resource utilization. But when we oversubscribe CPU further, the improvement diminishes and the result (e.g., the avg JCT of Y+S) starts to become worse.

The problem of simple over-subscription is *poor load balancing* and *resource contention*. First, although YARN maintains a good balance in container assignment, it does not guarantee load balancing. When resources in containers are not fully utilized, it leads to unbalanced loads among the workers. Specifically, for Y+S (and similarly for Y+U), the difference in the average CPU utilization rates among workers is 16%, 20% and 21% for a subscription ratio of 1, 2 and 4. In contrast, for Ursa the difference is only 2% (i.e., all workers have balanced, high CPU utilization).

Second, as each job assigns tasks to its allocated containers individually without knowing the cluster load, oversubscription leads to resource contention among jobs. Tasks on overloaded workers become stragglers, which renders containers on other workers under-utilized. We measured straggler as follows. We define the straggler threshold, following the general statistical definition of outliers, as the task completion time that is more than 1.5 times the interquartile range above the third quartile in the same stage. The straggler time for each stage is calculated as the completion time of the last task minus the threshold. We sum the straggler time of all stages for each job. The average ratio of the total straggler time to the JCT of all jobs in Y+U increases with the subscription ratio from 2.91% to 6.78% and 10.69%.

5.2 The Effects of Individual Designs

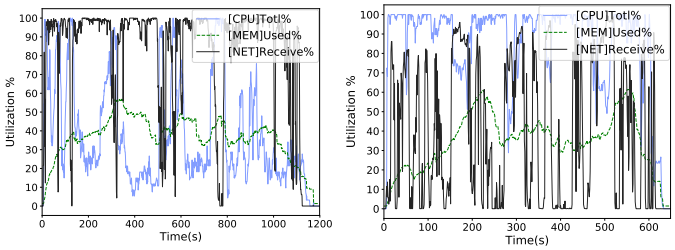
In this set of experiments, we analyze the effects of some individual designs in Ursa on its performance using the TPC-H2 workload.

The effects of network utilization. In modern clusters, network resource is quite sufficient for most workloads. In this case, can we ignore the network demands of tasks in task assignment? We compared the makespan and average JCT of Ursa, which are 650 and 383.25 seconds when we ignore network demands, and improved to 613 and 338.67 seconds when we consider network demands. Without considering network demands, monotasks with large network usage can be collocated and contend for network resource, which blocks their dependent CPU monotasks and hence impairs CPU utilization.

The above result is contrary to the use of Tetris in Ursa, which has better performance by ignoring the network demands as reported in §5.1.2 and Table 4. This is because Tetris uses peak resource demands, which is not a good measure for balanced task assignment as analyzed in §5.1.2. In contrast, *Ursa assigns tasks based on the overall resource usage of a task and the per-resource processing rate of a worker (§4.2.2), thus more effectively balancing the load of utilizing each resource among workers (the difference in the average network/CPU utilization among workers is 3.26%/2.93% only).*

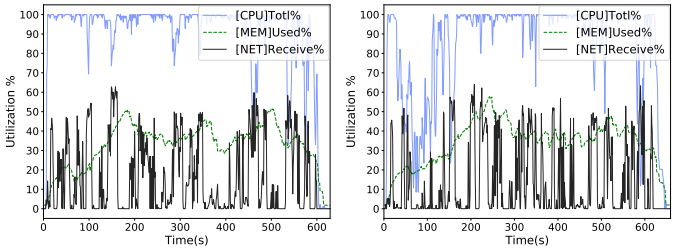
Is the design for high CPU utilization only? We have explained that Ursa does not have high utilization for network and memory because they are sufficient in our cluster. Now we validate this by limiting the network bandwidth between machines to 1Gbps and 4Gbps. With 1Gbps bandwidth, network becomes the bottleneck resource and as plotted in Figure 6a, Ursa achieves high network utilization, while CPU is not highly used as many CPU monotasks now need to wait for data from network monotasks that are busy contending the network bandwidth. When we increase the bandwidth to 4Gbps, the bottleneck resource starts to switch back to CPU and Figure 6b shows that CPU now becomes highly utilized and network utilization drops as a result of sufficient bandwidth. Comparing the resource utilization patterns of the 1Gbps, 4Gbps and 10Gbps networks in Figure 6a, 6b and 4a, we can see that *Ursa is not just optimized for CPU utilization but can generally achieve high utilization for the bottleneck resource. This also means that Ursa is suitable for both CPU-intensive and network-intensive workloads.*

The effects of stage-awareness. Ursa assigns tasks by stages. We modified the task assignment algorithm to pick one task (instead of a whole stage) that has the highest score each time. The makespan and average JCT increase by 5.66% and 10.84%



(a) 1Gbps **(b)** 4Gbps

Figure 6. Resource utilization (EJF) of different networks

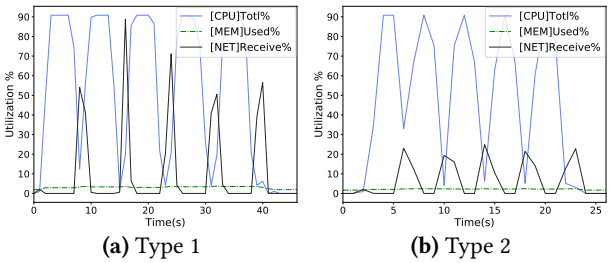


(a) Stage-aware **(b)** Non-stage-aware

Figure 7. (Non-)Stage-aware resource utilization (EJF)

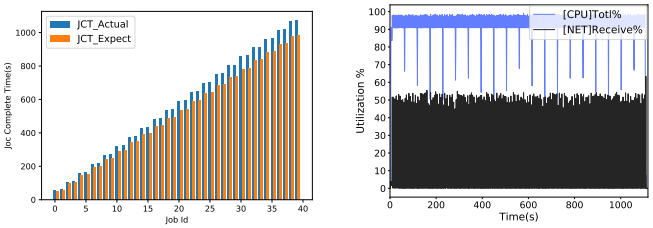
	makespan (EJF)	makespan (SRJF)	avg JCT (EJF)	avg JCT (SRJF)
JO	630.33	**623**	373.08	376.67
MO	615.33	629.33	351.73	346.49
JO + MO	**613**	635.67	**338.67**	**328.31**

Table 6. Results of job/task ordering (sec)



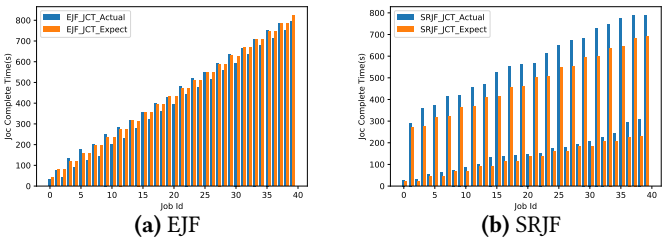
(a) Type 1 **(b)** Type 2

Figure 8. CPU and network utilization of 1 job



(a) 40 Type 1 jobs **(b)** Utilization

Figure 9. Setting 1: JCT and utilization



(a) EJF **(b)** SRJF

Figure 10. Setting 2: JCT

for EJF, and 10.28% and 15.73% for SRJF. The performance degradation can be explained as follows. When ready-to-run stages have similar resource demands and thus similar task scores, the tasks with lower scores in a stage are assigned after high-score tasks in other stages. Consequently, each stage has some low-score tasks remaining (i.e., stragglers) and blocks its dependent stages. When there are insufficient tasks to run as stages are being blocked, both CPU and network are under-utilized, as observed in the 30-180s period in Figure 7b. In contrast, *for stage-aware task assignment in Figure 7a, dependent stages are not blocked and CPU remains highly utilized* during the period.

The effects of job and task ordering. To evaluate how *job ordering* (*JO*) and *monotask ordering* (*MO*) contribute to the enforcement of EJF and SRJF in Ursa, we tested three settings: (1) JO only, (2) MO only, and (3) both JO and MO. Table 6 reports the makespan and average JCT for each setting, which shows that MO is more effective than JO in enforcing EJF and SRJF, but enabling EJF/SRJF for both JO and MO gives the best results. MO is more effective than JO because it directly determines both resource allocation and monotask execution. We also see, for JO+MO, SRJF gives worse makespan than EJF in exchange for better average JCT.

5.3 Experiments with Expectable Performance

For all the workloads we tested, due to the high heterogeneity of their jobs, it is difficult to expect what the optimal resource utilization, makespan and JCT are. We generated a synthetic workload for which we can calculate the ideal JCT and makespan, so as to evaluate whether Ursa can achieve the expected performance. We created synthetic jobs that consist of 5 stages (typical DAGs of depth 5 [19]). Each stage consists of homogeneous tasks that generate a large amount of random numbers and perform data shuffles. The parallelism of each stage is set as (30 cores × 20 machines). There are two types of jobs. Type 1 jobs handle twice the amount of data as Type 2 jobs.

When executed individually, the CPU and network utilization of the two types of jobs is reported in Figure 8, which shows regular fluctuations in both CPU and network utilization. The average JCT of Type 1 and Type 2 jobs are 40 and 22 seconds (roughly 8 and 4.4 seconds per stage). The

average CPU utilization for running a single job is 57% for Type 1 and 50% for Type 2.

In the first setting, we ran only Type 1 jobs using EJF. As each job uses CPU and network resources alternately, we expect Ursa to obtain the following result in the ideal case. According to the parallelism of each stage, the CPU monotasks of each job occupy all the 30 cores in each of the 20 machines. According to Algorithm 1, the tasks in one stage of a job will be assigned to workers each time. If Ursa can indeed enable fine-grained resource sharing among jobs, then we should see the following utilization pattern: *while the CPU monotasks of a job are using all CPU cores, the network monotasks of another job are using the network. And this pattern repeats immediately when a stage completes.*

Assume that the jobs are ordered by their submission timestamps as j_1 to j_{40}. By the EJF policy, the tasks of j_1 and j_2 will be assigned and executed first, and then followed by j_3 and j_4, and so on. As a result, we can calculate the expected JCT for each job in the ideal case, i.e., j_1 completes in 40 seconds, j_2 completes in (40 + 8) seconds, j_3 completes in (40 + 40) seconds, j_4 completes in (40 + 40 + 8) seconds, and so on. Figure 9a shows that the actual JCTs are very close to the expected JCTs in the ideal case, i.e., Ursa achieves near optimal JCT (and makespan). This verifies the effectiveness of Ursa's task placement and its efficiency in resource allocation and utilization. Figure 9b further shows that Ursa achieves stable, nearly full utilization of CPU, meaning that it can effectively address the problem caused by fluctuating resource utilization patterns.

Next in the second setting, we submitted 20 Type 1 and 20 Type 2 jobs alternately. This setting is harder to handle than the first setting as there is more variation in the resource utilization patterns. We tested the performance with both EJF and SRJF. Figure 10 shows that the actual JCT achieved by Ursa for both EJF and SRJF are close to the expected JCT in the ideal case. The CPU and network utilization patterns are also similar to Figure 9b and thus omitted.

6 Related Work

We discuss existing scheduling works on both system architecture designs and algorithms. For execution frameworks, we only discuss some closely related ones.

Schedulers. As discussed in §1, the architecture designs of existing schedulers fall into three categories: centralized, distributed, and hybrid. YARN [42], Mesos [21], and Borg [43] are (logically) centralized schedulers adopted in large clusters. In YARN[5] and Borg, a single scheduler makes scheduling decisions, while in Mesos scheduling decisions are made by individual applications and coordinated by a central resource provider. Sparrow [34], Tarcil [13], Apollo [3] are distributed schedulers designed for high scalability, in which multiple schedulers asynchronously assign tasks to workers based on their loads, and each worker maintains a local task queue. Sparrow uses batch-sampling to assign tasks to workers with lighter loads. Tarcil extends Sparrow by providing dynamic adjustment of sampling sizes based on the cluster load. Apollo maintains a wait-time matrix for CPU and memory on each worker for placing tasks. Yaq-c [35] describes an architecture where a centralized scheduler places tasks to worker queues, which is most similar to Ursa's architecture. However, Yaq-c adopts slot-based resource allocation and uses historical task execution time to estimate waiting time at workers for task placement. Thus, Yaq-c does not satisfy Obj-1&2 for processing workloads with dynamic resource usage. Hawk [10], Eagle [9] and Mercury [26] are hybrid schedulers, where critical tasks (i.e., tasks with high priority or long running time) are scheduled by a centralized scheduler, while other tasks are allocated by distributed schedulers.

Many algorithms have been proposed to achieve different scheduling objectives, including fairness [14, 24, 49], SLO [11, 12, 41], resource utilization [8, 11, 12, 15, 17, 24, 31], and JCT [14, 18, 19, 25, 27]. Ursa focuses on improving resource utilization by more accurate and fine-grained resource allocation for dynamic workloads.

Execution frameworks. Dataflow model is commonly adopted in general-purpose data analytics frameworks [4, 23, 36, 46, 50]. Apache Hive [39] is a widely adopted data warehouse system that supports OLAP. It provides query optimization and processing, and allows plugin of different execution engines including Spark and Tez. Ursa has a connector to Hive for SQL execution.

MonoSpark [33] extends Spark for performance clarity of jobs. It monitors queues of monotasks at each server for reliable performance reasoning based on per-resource usage. Thus, the use of monotask in [33] is different from our work, as Ursa enables fine-grained resource sharing among jobs by attaching resource allocation to monotask execution and provides runtime resource demands based on monotasks. We used Ursa to simulate MonoSpark in §5.1.2.

7 Conclusions

We presented Ursa — a framework for both resource scheduling and job execution. Ursa is designed to handle jobs with frequent fluctuations in resource usage, which are commonly found in workloads such as OLAP, machine learning and graph analytics. Our experimental results validate that Ursa's designs are effective in achieving high resource utilization, which is translated into significantly improved makespan and average JCT.

Acknowledgments. We thank the reviewers and shepherd Dan Tsafrir of our paper for their constructive comments that have helped improve the quality of the paper. This work was supported in part by ITF 6904945 and GRF 14208318.

[5] Since YARN 2.9.2, a distributed scheduler has been added to handle opportunistic containers.

References

[1] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2019. *A Large Time-Aware Graph.* http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05/

[2] Léon Bottou. 2019. *MNIST8M - The infinite MNIST dataset.* https://leon.bottou.org/projects/infimnist

[3] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 285–300.

[4] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[5] Carlos Castillo. 2019. *Datasets for Research on Web Spam Detection.* http://chato.cl/webspam/datasets/

[6] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. 2016. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 407–424.

[7] Cloudera. 2015. *How-to: Tune Your Apache Spark Jobs.* https://blog.cloudera.com/how-to-tune-your-apache-spark-jobs-part-2/

[8] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 17)*. ACM, 153–167.

[9] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2016. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC 16)*. ACM, 497–509.

[10] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. 2015. Hawk: Hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 499–510.

[11] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 13)*, Vol. 48. ACM, 77–88.

[12] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 14)*, Vol. 42. ACM, 127–144.

[13] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. 2015. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC 15)*. ACM, 97–110.

[14] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types.. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Vol. 11. 24–24.

[15] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. 2016. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 99–115.

[16] Google. 2019. *Production-Grade Container Orchestration - Kubernetes.* https://kubernetes.io/

[17] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2015. Multi-resource packing for cluster schedulers. In *Proceedings of the ACM SIGCOMM Computer Communication Review (SIGCOMM 15)*, Vol. 44. ACM, 455–466.

[18] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic scheduling in multi-resource clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 65–80.

[19] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 81–97.

[20] Apache Hadoop. 2019. *Hadoop: Capacity Scheduler.* https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html

[21] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center.. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Vol. 11. 22–22.

[22] Yuzhen Huang, Yingjie Shi, Zheng Zhong, Yihui Feng, James Cheng, Jiwei Li, Haochuan Fan, Chao Li, Tao Guan, and Jingren Zhou. 2019. Yugong: Geo-Distributed Data and Job Placement at Scale. *PVLDB* 12, 12 (2019), 2155–2169. https://doi.org/10.14778/3352063.3352132

[23] Yuzhen Huang, Xiao Yan, Guanxian Jiang, Tatiana Jin, James Cheng, An Xu, Zhanhao Liu, and Shuo Tu. 2019. Tangram: Bridging Immutable and Mutable Abstractions for Distributed Data Analytics. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 191–206. https://www.usenix.org/conference/atc19/presentation/huang

[24] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 09)*. ACM, 261–276.

[25] Prajakta Kalmegh and Shivnath Babu. 2019. MIFO: A Query-Semantic Aware Resource Allocation Policy. In *Proceedings of the 2019 ACM International Conference on Management of Data (SIGMOD 19)*. ACM, 1678–1695.

[26] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 485–497.

[27] James E Kelley Jr and Morgan R Walker. 1959. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*. ACM, 160–173.

[28] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop.. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems (CIDR 15)*, Vol. 1. 9.

[29] Kubernetes. 2015. *Taking network bandwidth into account for scheduling.* https://github.com/kubernetes/kubernetes/issues/16837

[30] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 14)*. ACM, 1–15.

[31] Libin Liu and Hong Xu. 2018. Elasecutor: Elastic Executor Scheduling in Data Analytics Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 18)*. ACM, 107–120.

[32] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB 10)* 3, 1-2 (2010), 330–339.

[33] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for performance clarity in

data analytics frameworks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 17)*. ACM, 184–200.

[34] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP 13)*. ACM, 69–84.

[35] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. 2016. Efficient queue management for cluster scheduling. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys 16)*. ACM, 36.

[36] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. 2015. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM International Conference on Management of Data (SIGMOD 15)*. ACM, 1357–1369.

[37] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys 13)*.

[38] Xiaoyang Sun, Chunming Hu, Renyu Yang, Peter Garraghan, Tianyu Wo, Jie Xu, Jianyong Zhu, and Chao Li. 2018. ROSE: Cluster Resource Scheduling via Speculative Over-Subscription. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS 18)*. IEEE, 949–960.

[39] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB 09)* 2, 2 (2009), 1626–1629.

[40] TPC-H. 2019. *Decision support benchmark*. http://www.tpc.org/tpch/

[41] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys 16)*. ACM, 35.

[42] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Saha Bikas, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing (SoCC 13)*. ACM, 5.

[43] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys 15)*. ACM, 18.

[44] Markus Weimer, Yingda Chen, Byung-Gon Chun, Tyson Condie, Carlo Curino, Chris Douglas, Yunseong Lee, Tony Majestro, Dahlia Malkhi, Sergiy Matusevych, Brandon Myers, Shravan M. Narayanamurthy, Raghu Ramakrishnan, Sriram Rao, Russell Sears, Beysim Sezgin, and Julia Wang. 2013. Reef: Retainable evaluator execution framework. *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB 13)* 6, 12 (2013), 1370–1373.

[45] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data* 1, 2 (2015), 49–67.

[46] Fan Yang, Jinfeng Li, and James Cheng. 2016. Husky: Towards a More Efficient and Expressive Distributed Computing Framework. *PVLDB* 9, 5 (2016), 420–431. https://doi.org/10.14778/2876473.2876477

[47] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.

[48] Yi Yao, Han Gao, Jiayin Wang, Ningfang Mi, and Bo Sheng. 2016. OpERA: opportunistic and efficient resource allocation in Hadoop YARN by harnessing idle resources. In *Proceedings of the 25th International Conference on Computer Communication and Networks (ICCCN 16)*. IEEE, 1–9.

[49] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems (EuroSys 10)*. ACM, 265–278.

[50] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, 2–2.

[51] Xiaoda Zhang, Zhuzhong Qian, Sheng Zhang, Xiangbo Li, Xiaoliang Wang, and Sanglu Lu. 2018. COBRA: Toward Provably Efficient Semi-Clairvoyant Scheduling in Data Analytics Systems. In *2018 IEEE Conference on Computer Communications (IEEE INFOCOM)*. IEEE, 513–521.

[52] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 301–316.