UBIS: Utilization-aware cluster scheduling

Karthik Kambatla
Facebook
kasha@fb.com

Vamsee Yarlagadda
Cloudera
vamsee@cloudera.com

Íñigo Goiri
Microsoft Research
inigog@microsoft.com

Ananth Grama
Purdue University
ayg@cs.purdue.edu

***Abstract*—Data center costs are among the major enterprise expenses, and any improvement in data center resource utilization corresponds to significant savings in true dollars. We focus on the problem of scheduling jobs in distributed execution environments to improve resource utilization. Cluster schedulers like YARN and Mesos base their scheduling decisions on resource requirements provided by end users. It is hard for end-users to predict the exact amount of resources required for a task/ job, especially since resource utilization can vary significantly over time and across tasks. In practice, users pick highly conservative estimates of peak utilization across all tasks of a job to ensure job completion, leading to resource fragmentation and severe under utilization in production clusters. We present UBIS, a utilization-aware approach to cluster scheduling, to address resource fragmentation and to improve cluster utilization and job throughput. UBIS considers actual usage of running tasks and schedules opportunistic work on under-utilized nodes. It monitors resource usage on these nodes and preempts opportunistic containers when over-subscription becomes untenable. In doing so, UBIS utilizes wasted resources while minimizing adverse effects on regularly scheduled tasks. Our implementation of UBIS on YARN yields improvements of up to 30% in makespan for representative workloads and 25% in individual job durations.**

***Index Terms*—distributed computing; scheduling; cluster schedulers**

I. Introduction

Modern cluster schedulers, like YARN [20], support a resource-request model, where users (via applications or frameworks) can request a specified amount of resources like CPU and memory. The requested resources are allocated when they become available. Availability is determined by the aggregate of cluster resources, less the sum of all prior allocations. YARN refers to these allocations as *containers*; henceforth also referred to as *regular containers*. The resources of a container are reserved exclusively for its use and cannot be used by other containers even if they are not used by the owning container. This leads to resource wastage.

To limit this resource wastage, users must request tight (over)estimates of required resources. In practice though, it is hard to accurately estimate the resource requirements of a job or its constituent tasks because: (i) resource usage of a task varies over time, and (ii) resource usage can vary across tasks of the same job based on the input they process. Users are expected to estimate and request the peak usage across all tasks to ensure job completion. This problem is further exacerbated by the fact that end-users use convenience wrapper libraries like Apache Hive [2] to create a majority of these jobs and are consequently unaware of their characteristics. For these reasons, users often rely on defaults, picking conservative estimates of peak utilization, or using requirements from other work-flows that are known to work. Over-provisioning tasks in this manner can lead to severe under-utilization in clusters [5, 7]. Figure 1 shows resource allocation and utilization on a 2200-node production YARN cluster at Yahoo! over a week. The figure shows that both memory and CPU usage are consistently under 50%, even when the cluster is fully allocated — the area between solid and dotted lines corresponds to resources allocated but not used. This typical resource utilization profile motivates our effort.

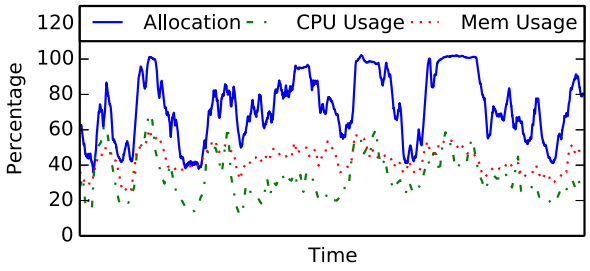


Fig. 1: Allocation and utilization in a 2200-node production YARN cluster at Yahoo!

In this paper, we present UBIS (utilization-based incremental scheduling), a scheduling approach that considers both user-specified requests and the actual utilization of previously allocated containers. Once the cluster is fully allocated (i.e., no additional tasks can be scheduled based on un-allocated cluster resources), UBIS opportunistically allocates resources not utilized by prior allocations. We refer to these allocations as *opportunistic containers*. These opportunistic containers use slack in the cluster to improve cluster utilization and job throughput. Oversubscribing node and cluster resources in this manner poses challenges. Oversubscription can become untenable when tasks simultaneously start using more resources, potentially leading to performance degradation and task failures. UBIS preempts opportunistic containers to ease any resource contention, but note that these preemptions limit throughput gains from opportunistic scheduling.

Improvements in resource utilization through oversubscription must not interfere with other desirable features of cluster schedulers, such as fairness. Pareto-efficient weighted-fairsharing [1, 9] enables equitable sharing of resources among multiple tenants[1]. The notion of *opportunistic resources*, where all tenants are interested in regular resources but only a subset of tenants are interested in opportunistic resources,

[1]Pareto efficiency describes a fully allocated cluster where it is impossible to assign a tenant more resources without adversely affecting another tenant.

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00045

introduces an additional dimension of fairness, making it harder to reason about overall fairness. UBIS addresses this by tracking fairness for regular and opportunistic resources separately: regular resources are fairly allocated to tenants, followed by a fair allocation of opportunistic resources among the subset of tenants that can tolerate opportunistic containers. Finally, owing to the scale and complexity of typical deployments, it is unrealistic to expect any runtime inputs from end-users. Any improvements must be transparent to users, preferably requiring no modifications to the user jobs themselves.

We present an implementation of UBIS in Apache YARN. Cluster administrators can turn on UBIS on a per-node basis to oversubscribe resources at the node and control scheduling through two knobs for allocating and preempting opportunistic tasks respectively. While most batch-processing jobs can tolerate preemptions in lieu of potential throughput gains, certain latency-sensitive applications might not be able to tolerate preemptions. These jobs can opt out of opportunistic allocations.

Our contributions can be summarized as follows:

1) Techniques for opportunistic allocation of un-utilized resources for improved utilization and job throughput, along with (1) graceful handling of untenable oversubscription through preemption of opportunistic containers, and (2) configuration knobs to control aggressiveness of opportunistic scheduling.
2) Notion of multi-dimensional fairness to accommodate resources of multiple priorities in a cluster.
3) Implementation of the techniques in Apache YARN, demonstrating improvements of up to 30% in makespan[2] for a representative workload and 25% in individual job durations.

II. Problem Motivation

In this section, we examine sample YARN workloads to investigate the source of under-utilization and make a case for utilization-aware scheduling.

A. Resource wastage at the job level

We select wordcount and the *tera-suite* of jobs – TeraGen, TeraSort, and TeraValidate for profiling. These jobs mimic common data-access patterns and are commonly used for performance comparisons. Wordcount counts the number of occurrences of each word in input data, and represents a variety of applications that compute statistics for given input – logs, clickstreams etc. Teragen is a map-only job, where each task generates some random data. Terasort sorts the data generated by TeraGen, and TeraValidate validates that the output of TeraSort is indeed sorted. The Tera-suite of jobs is representative of an ETL (extract-transform-load) pipeline.

Figure 2 shows the memory usage for sample runs. All jobs request default 1 GB containers. For each job, the figure plots: (i) the *overall mean*, defined as the mean of all per-container mean usages, (ii) the *mean peak*, defined as the mean of all per-container peak usages, and (iii) the *overall peak*, defined as the max of all per-container peak usages. The overall peak is the least amount of memory that can be requested; requesting less memory will lead to task and job failures. From the plots, we make the following observations:

[2]Makespan is the total execution time for the workload.

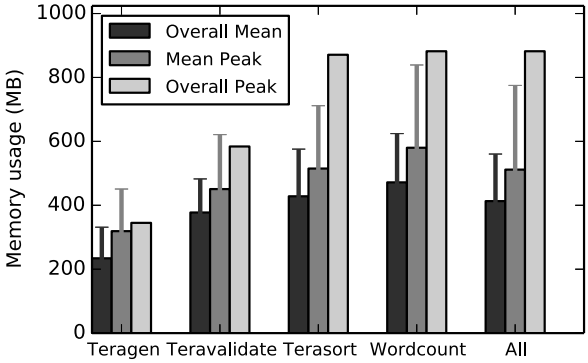


Fig. 2: Memory usage for commonly used Hadoop examples

1) The *mean peak* usage is noticeably higher than the *overall mean*. This is because the resource usage of a task varies over its different stages — read input, process, and write output — each stage with a different resource usage profile.
2) The error bars in the figure for both *overall mean* and *mean peak* usage show that resource usage can vary significantly across tasks of the same job.
3) The *overall mean* is roughly half the *overall peak*, showing that only half of a container's resources are typically used.

These results show that under-utilization can be significant, even if users submitting these jobs are sophisticated enough to track the peak usage of constituent tasks.

B. Resource wastage in workflows

Enterprise users typically use convenience libraries (like Hive) to submit workflows (SQL queries), posing additional challenges. Since the constituent jobs are created at runtime, even a sophisticated user cannot specify the peak resource usage for each job in the workflow. Hive allows specifying resource requirement for the entire workflow, requiring the user to specify the peak usage across all tasks of all jobs that are part of the workflow. However, enterprise users are often unaware of constituent jobs, let alone their resource usage. It is common for these users to use conservative defaults these software packages ship with, pick values by trial-and-error for which the query succeeds, or even copy values that worked for another query that is potentially more resource intensive.

One way to lower this under-utilization is to improve workflow systems to better estimate the resource requirements of constituent jobs. For instance, prior work [8] estimates the duration of constituent jobs, and a similar approach may apply to estimating resource requirements. Improved estimates will still be a tight bound on the peak usage, which could be significantly higher than the mean usage as discussed earlier.

III. Problem Statement

UBIS aims to improve effective cluster utilization, which directly translates to metrics that impact end-users: makespan of workloads and individual jobs. For a given workload, makespan is the duration between workload submission and completion. A workload can be viewed as a directed acyclic graph (DAG) of jobs, where each job comprises several tasks. A single job is the basic unit of a workload, and makespan for a job is simply the job duration.

Consider a cluster with R resources (R is a multidimensional vector, with dimensions representing CPU, memory, and potentially other resources), and a workload comprising of N jobs with a total of n tasks. We denote by R_i the maximum resource requirement of task T_i, as specified by the user. Note that R_i can be computed by maximizing along each dimension over the execution of the task. At any given time, the scheduler schedules the largest subset of tasks that fit on the cluster, $G = \{T_i | \sum R_i \leq R\}$; adding one more task to the set would lead to $\sum R_i > R$. We represent by U_i the actual utilization of task T_i running on the cluster. As shown in §II, $U_i < R_i$, and $R^s = \sum R_i - \sum U_i$ captures the slack in the cluster due to under-utilization.

UBIS proposes to schedule opportunistic tasks to use this slack, R^s. One could allocate the largest subset of tasks that fit in this slack, $O = \{T_i | \sum R_i < R^s\}$. However, utilizing all available slack can cause resource contention (with possible task failures) due to temporal variations in resource utilization. The associated loss in performance is captured by Δt_i, where t_i is the duration of task T_i. Under extreme contention, a select number of tasks (p) might need to be preempted to ensure that oversubscription remains tenable. We make the following observations:

1) Makespan is inversely related to the number of tasks run in parallel.
2) Makespan grows with task duration ($t_i + \Delta t_i$).
3) Δt_i grows with number of parallel tasks due to resource contention. An unsustainable number of parallel tasks may lead to thrashing.
4) Makespan grows with number of preemptions (p).

Optimal makespan corresponds to the largest value of utilized slack for which resource contention is manageable; i.e., $\sum \Delta t_i$ and p remain small. This depends on the workload, as well as cluster resources. Further, improvements in utilization and makespan must not interfere with other aspects of cluster scheduling. Our design of UBIS has the following considerations:

C1 The cluster should remain operational; none of the nodes should fall over due to the additional load of opportunistic containers.

C2 Job durations should be predictable and at least on par with the case of no opportunistic containers.

C3 Resource allocation should continue to honor fairness requirements, as outlined in [9], for both regular and opportunistic allocations.

C4 Jobs should be allocated regular containers no later than the base case of no opportunistic containers.

C5 Scalability of the scheduler should not be affected.

C6 Effect of opportunistic scheduling on the execution of regular containers should be minimal.

C7 Cluster administrators should be able to turn on UBIS without the need for any end-user action.

Constraints C1 - C5 are essential for clusters to adopt UBIS. C3, C4 and C5 capture scheduling requirements, while C1 and C2 capture requirements on the execution environment. C6 and C7 are desirable and make for good user experience, but are not essential.

IV. UBIS Design

In this section, we outline our proposed solution to the underlying optimization problem, and discuss how our solution addresses constraints listed in the previous section.

A. Identifying the opportunity

UBIS identifies resource slack at each node by actively monitoring resource usage of each container. UBIS augments the node heartbeat [3] to include utilization information as well as an *over-allocation threshold* (T_{alloc}); T_{alloc}, a value between 0 and 1, specifies the extent of oversubscription allowed on that node. The scheduler allocates opportunistic containers only if utilization is less than $T_{alloc} \times R_n$, where R_n is the node resource capacity for running containers.

B. Scheduling opportunistic containers

In both Yarn and Mesos, resources are allocated to nodes on node heartbeats. When a node heartbeats, (1) the scheduler updates its state (resource availability and container state changes) and (2) allocates containers by iterating through waiting jobs, in the order determined by fairness constraints, and checking if the node meets the resource and locality requirements. The scheduler continues to allocate containers to the node, as long as it has enough resources to meet a job's pending request.

Algorithm 1 presents the UBIS scheduling algorithm. UBIS schedules regular containers the same way the base scheduler does (lines 11 to 19) by calling *AllocateRegularContainer* as long as there are enough unallocated resources to meet a pending request. Once the scheduler allocates regular containers, it schedules opportunistic containers (lines 21 to 29) by calling *AllocateOpportunisticContainer* as long as there are enough un-utilized resources on the node to meet a pending request. *AllocateOpportunisticContainer* differs from *AllocateRegularContainer* only in how the node's availability is computed; resource availability for opportunistic containers is computed based on node utilization ($Util$) and T_{alloc} (line 21). Note that the scheduler assumes the newly allocated containers will utilize all allocated resources; this headroom accommodates minor fluctuations in usage of running containers.

[3] In most cluster schedulers, worker nodes heartbeat container liveness information to the master periodically.

Algorithm 1: UBIS scheduling for a node, N

Result: Container allocations for the node

$Cap \leftarrow GetNodeCapacity(N)$
$Alloc^{reg} \leftarrow GetRegularAllocation(N)$
$Util \leftarrow GetUtilization(N)$
$T_{alloc} \leftarrow GetOverAllocThreshold(N)$

foreach *opportunistic container* C **do**
    PromoteIfPossible(C)
end

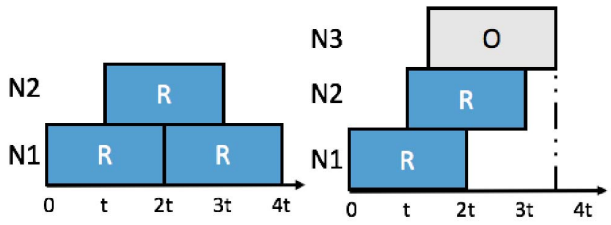
$Alloc^{new} \leftarrow 0$
$Avail^{reg} \leftarrow Cap - Alloc^{reg}$
while $Alloc^{new} < Avail^{reg}$ **do**
    $Alloc^{tmp} \leftarrow AllocateRegularContainer(N)$
    if $Alloc^{tmp}$ *is valid* **then**
        $Alloc^{new} \leftarrow Alloc^{new} + Alloc^{tmp}$
    else
        break
    end
end

$Avail^{opp} \leftarrow T_{alloc} \times Cap - Util$
while $Alloc^{new} < Avail^{opp}$ **do**
    $Alloc^{tmp} \leftarrow AllocateOpportunisticContainer(N)$
    if $Alloc^{tmp}$ *is valid* **then**
        $Alloc^{new} \leftarrow Alloc^{new} + Alloc^{tmp}$
    else
        break
    end
end

C. Avoiding adverse effects of opportunistic containers

The addition of opportunistic containers for improved utilization can lead to contention, sometimes not allowing the node agent (or OS) to react with a corrective action, thereby rendering the node unusable. These concerns correspond to the constraints C1 and C2 mentioned in §III.

To avoid these adverse conditions, UBIS proposes that the node agent proactively preempt opportunistic containers to avoid severe contention. The effects of contention vary for different resources: memory contention can lead to task failures, whereas CPU contention causes performance degradation for low contention; only high contention with unsustainable context-switch overhead leads to failures. For malleable resources like CPU, we can tolerate 100% utilization for short bursts of time. Accordingly, UBIS introduces a per-resource *preemption-threshold*, $T_{preempt}$, (a value between 0 and 1); if the aggregate container utilization goes beyond $T_{preempt}$ for a preset number of heartbeats, the node agent preempts enough opportunistic containers to bring the utilization under the threshold. Note that the scheduler does not play an active role in this preemption to ensure responsiveness, but follows up with appropriate allocations when notified in a subsequent heartbeat.



(a) Regular (b) Regular + Opportunistic

Fig. 3: Task schedule and execution for a job with 3 tasks. X-axis plots time and Y-axis represents nodes N1, N2, and N3.

UBIS also places certain safeguards in addition to proactive preemption. Opportunistic containers are run at a lower priority than regular containers, as allowed by the operating system and the execution environment. On Linux, we leverage the host operating system through cgroups to limit the aggregate container utilization to the configured threshold. Cgroups may also be used to isolate regular containers by offering malleable resources (CPU, disk, and network) only on a best-effort basis to opportunistic containers. If opportunistic containers are consistently deprived, they are preempted to be scheduled elsewhere, potentially as regular containers.

D. Promotions and opt-out provisions

The effect of allocating opportunistic containers, occasionally at the expense of delayed regular containers, depends on the workload. Most jobs do not have service level agreements (SLAs) and increased parallelism helps improve makespan. Consider the job execution illustrated in Figure 3. The job has three tasks, each task runs for $2t$ units of time when run as a regular container. Without opportunistic containers (Figure 3a), a new regular task is scheduled every t units of time and the job takes $4t$ units of time to finish. With opportunistic containers enabled (Figure 3b), the third task is scheduled opportunistically on node N3 at time $1.5t$, ahead of the regular container allocation at $2t$. During the time interval $2t$ to $3t$, the job has fewer regular containers than the base case, violating constraint C4. Despite this violation, this particular job benefits from the opportunistic allocation finishing at time $3.5t$, even though the opportunistic task takes slightly longer than the regular task.

To limit C4 violations, UBIS attempts to *promote* opportunistic containers to regular containers. Same node promotions are easy and always beneficial; the scheduler and the worker node need to update their book-keeping and adjust any isolation settings. Cross node promotions are more involved: if the job does not have a way to checkpoint and migrate the task in a work-preserving manner, the task must be killed restarted on another node. As a result, cross-node promotion may lead to longer runtime than letting the opportunistic container run. Since common frameworks (MapReduce and Spark) do not preserve work across task restarts, UBIS does not promote containers across nodes. If a job's SLA requirements are tight and same node promotions alone are not enough to guarantee them, UBIS allows these jobs to *opt out* of opportunistic allocations. The number of jobs that fall into this category

is typically small.

Note that jobs may also adopt a hybrid approach. Instead of categorically opting out of opportunistic allocation, they may choose to not accept opportunistic containers at specific points of execution. For instance, a MapReduce job with large shuffle data might not want to run reduce tasks in opportunistic containers, since the cost of preemption is high; on the other hand, opportunistic containers are well-suited to speculative tasks.

E. Honoring fairness

Fairness-based schedulers [1, 9, 10] are widely used to share resources among cluster users. Ghodsi et al.[9] outline desirable characteristics of a fairness-based cluster scheduler and show that Dominant Resource Fairness (DRF) honors each of these characteristics:

Sharing incentive. Users should have an incentive to share a cluster; if there are n users, each user should be allocated at least $\frac{1}{n}$ of all resources in the cluster. Otherwise, they might be better off running their own partition.

Strategy-proof. A user should not get a larger allocation by misrepresenting her resource requirements. For example, a user should not get more resources just by asking for larger containers.

Envy-free. A user should not prefer the allocation of another user.

Pareto-efficiency. On a fully allocated cluster, a user cannot be assigned more resources without adversely affecting another user's allocation.

The Hadoop FairScheduler [1] implements both max-min fairness and DRF, and the user can choose one of them. UBIS builds on the fairness of the base scheduler by applying the base scheduling algorithm to allocate regular and opportunistic containers. Allocation of regular containers is identical to the base case: unallocated cluster resources are fairly allocated among interested tenants. Opportunistic containers are allocated only when the scheduler cannot allocate any more regular containers. Allocation of opportunistic containers applies the base scheduling algorithm to the pool of un-utilized resources, instead of unallocated resources, leading to a fair allocation of un-utilized resources among interested tenants. If a job opts out of opportunistic containers, it is not allocated any un-utilized resources and its *share* is distributed among other jobs.

Since UBIS applies DRF to unallocated/ un-utilized resources to allocate regular and opportunistic containers respectively, it satisfies each of these characteristics.

Theorem 1. UBIS satisfies sharing incentive property.

Proof. Assume that UBIS does not satisfy the *sharing incentive* property. On a cluster with n users, a user's total allocation is less than $\frac{1}{n}$ of all resources. At least one of regular or opportunistic allocation must be less than $\frac{1}{n}$ of the corresponding resources available.

Case 1. User's regular allocation is less than $\frac{1}{n}$ of regular resources. Since regular resources are allocated by applying DRF, this means DRF does not satisfy sharing incentive property, which contradicts our assumption.

Case 2. User's opportunistic allocation is less than $\frac{1}{n}$ of opportunistic resources. If the user has opted out of opportunistic allocations, this is expected; even on a partition with $\frac{1}{n}$ of cluster resources, the user would not be allocated any opportunistic containers. If the user has not opted out of opportunistic allocations, again the allocation was by applying DRF, and this contradicts our assumption that DRF satisfies sharing incentive. □

Similarly, building on the proofs in [9], one can show by contradiction that UBIS honors all the listed characteristics. In fact, UBIS incentivizes sharing the cluster more than plain DRF. In addition to unallocated resources, users sharing the cluster are allocated opportunistic containers when other users are not fully utilizing their regular allocations. UBIS drives the utilization higher by being pareto-efficient in both regular and opportunistic resources.

F. Scalability

Scheduler scalability depends on: (i) in-memory state associated with nodes, jobs, containers, and (ii) handling node heartbeats and allocating containers on these nodes (serialization at the scheduler). UBIS does not alter the number of nodes, jobs, or heartbeats. UBIS, however, allocates more containers than the base case: this does not impede scalability from a processing perspective, since all these containers would be allocated at some point, but does lead to a marginally larger memory footprint. UBIS also augments the node heartbeat to include node utilization information; this leads to a slight increase in network traffic and time for processing node heartbeats. In practice, the overhead of these changes is marginal. From our experience developing and supporting YARN on production clusters, this is quite manageable.

V. Evaluation

We provide the necessary background on YARN and outline the changes we made to realize UBIS in YARN. Our evaluation of this implementation is two-fold: (i) sensitivity analysis aimed at understanding the effect of different factors in a controlled environment; and (ii) real-world analysis running UBIS on a 20 node cluster running a workload representative of typical production workloads to quantify performance improvements.

A. Implementation

Background. YARN currently supports CPU and memory as resources[4]. YARN ships with three schedulers – FairScheduler, CapacityScheduler and FifoScheduler – the first two are commonly used. We implemented the UBIS scheduling algorithm (Algorithm 1) in both FairScheduler and CapacityScheduler, but limit our discussion to FairScheduler. We refer to the

[4]YARN has recently added support for scheduling generic resources, but the node agents track only CPU and memory.

Tab. I: Workload for sensitivity analysis

Job	Task memory usage
Sleep	Fixed 200 MB
G-Sleep-50	Up to 50% of allocated heap
G-Sleep-75	Up to 75% of allocated heap
G-Sleep	Up to 99% of allocated heap

existing implementation as the *base* implementation and the augmented version as the *UBIS* implementation.

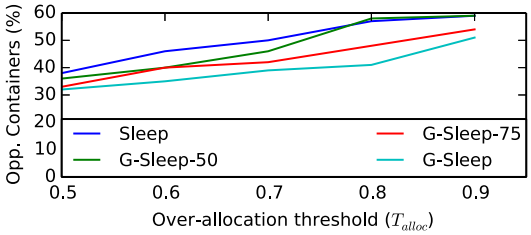
Collecting resource utilization. The NodeManager (YARN's node agent) already monitors the CPU and memory usage of individual containers using the *proc* filesystem on Unix-based operating systems. We augmented this to compute the aggregate CPU and memory usage across all YARN containers on the node. We send this information to the scheduler via the periodic node heartbeat to include: (i) aggregate container resource utilization and (ii) over-allocation threshold (T_{alloc}). This additional information — memory and CPU utilization are an integer and a float, respectively, and T_{alloc} is a float for each resource — adds little overhead to the node heartbeat payload.

Allocating opportunistic containers. On receiving the node heartbeat, our UBIS implementation allocates regular containers the same way the base implementation does. After scheduling regular containers, the scheduler allocates opportunistic containers to use the resources that are allocated to other containers, but are not being actively utilized. Note that the containers being allocated in this heartbeat are yet to be started and hence have zero utilization at the time of allocation. In our implementation, we conservatively assume that these newly allocated containers will use all the resources allocated to them. So, the effective resource availability on a node for opportunistic allocations is its capacity, less last reported utilization and prior allocations in this heartbeat ($capacity - actual\_utilization - prior\_allocations$). This conservative estimate helps us avoid situations in which all containers start simultaneously and overload the worker node.

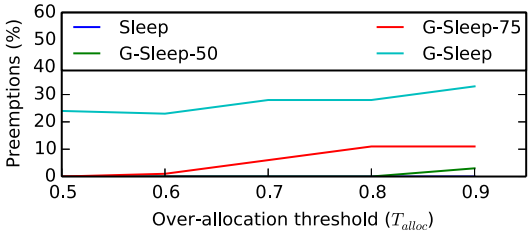
Keeping oversubscription viable on the node. In the event of tasks simultaneously using more resources, over-allocation could lead to severe contention and potential task failures. UBIS proposes the use of a per-resource $T_{preempt}$; we implemented this in YARN for CPU and memory. If the resource usage goes over this threshold, we preempt enough opportunistic containers to control resource contention. For a malleable resource like CPU, we tolerate brief spikes and preempt containers only on sustained contention. To accommodate this, we add another tunable parameter called *preemption-threshold-count*; containers are preempted only if the contention sustains for *preemption-threshold-count* number of aggregate container usage checks. By default, an aggregate container usage check is triggered every 1 second and the *preemption-threshold-count* is set to three. These parameter defaults have been inferred from empirical observations of real environments.

B. Sensitivity Analysis

Our sensitivity analysis is aimed at understanding the effect of various parameters on the number of opportunistic allocations and preemptions, and any workload characteristics that determine the extent of improvement from this opportunistic scheduling. For this analysis, we use a modified version of the MapReduce sleep job. The sleep job runs a specified number of map and reduce tasks, each of which sleep for a specified duration. These sleep tasks use minimal resources — 200 MB for the JVM and small fraction of a core. We modified this job to gradually grow task memory usage to a specified limit, by allocating memory on the heap. Table I captures the four variations of the modified sleep job we use in this analysis. We ran the sensitivity analysis experiments on a single node cluster with 72 physical cores and 256 GB RAM. Of this, 64 cores and 130 GB memory are allotted to YARN containers. Each job has 315 map tasks, each running on a ⟨1 core, 2 GB⟩ container and 0 reduce tasks. In a MapReduce job, one container (1 core) is used by the ApplicationMaster[5], leaving 63 cores for the tasks themselves. We chose 315 map tasks per job so each job has 5 waves of map tasks.



(a) Effect on opportunistic allocations.



(b) Effect on preemptions.

Fig. 4: Effect of T_{alloc} on opportunistic allocations and preemptions. $T_{preempt}$ = 0.9f. Sleep task duration = 1 min.

Note that even though we tightly control the resource usage of containers for this sensitivity analysis, the order of container resource requests and allocations is non-deterministic and can affect scheduler allocations of regular and opportunistic containers. The relative timings of container starts can also affect actual aggregate utilization, and therefore the number of preemptions due to resource starvation. To limit the effect of this variability, we run each job five times serially and use the mean values for all metrics we compute for our conclusions.

Effect of over-allocation threshold (T_{alloc}). Figure 4 captures the effect of T_{alloc} on: (i) the percentage of opportunistic containers allocated relative to the total number of containers

[5]In Yarn, the ApplicationMaster negotiates resources with the scheduler and launches tasks for the application (job) on worker nodes.

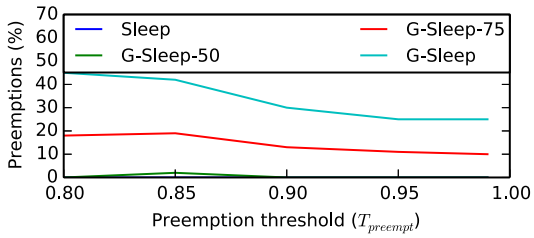


Fig. 5: Percentage of preemptions at different values of $T_{preempt}$. T_{alloc} = 0.8f. Sleep task duration = 1 min.

(Figure 4a); and (ii) the percentage of opportunistic containers subsequently preempted due to resource contention relative to the total number of containers (Figure 4b).

Jobs with low per-task utilization (more unused resources) have a higher percentage of opportunistic containers; the vanilla Sleep job has the highest percentage of opportunistic containers. The percentage of opportunistic containers grows with T_{alloc}; the percentage of opportunistic containers increases from $30\% - 40\%$ at $T_{alloc} = 0.5$ to $48\% - 60\%$ at $T_{alloc} = 0.9$. The rate of growth is lower at higher values of T_{alloc}. Jobs with greater variability in utilization have more preemptions; the preemption percentage for G-Sleep is highest, and the vanilla Sleep job has no preemptions. As containers use more resources, the earlier value for utilization used by the scheduler becomes an underestimate, leading to resource contention and preemptions. T_{alloc} only marginally affects the rate of preemptions. In Figure 4b, the increase in preemptions between T_{alloc} values 0.5 and 0.9 is less than 10% of total containers.

Effect of preemption threshold ($T_{preempt}$). Figure 5 captures the effect of $T_{preempt}$ on the percentage of opportunistic containers preempted relative to the total number of containers. As expected, the rate of preemptions goes down as we increase $T_{preempt}$. A wider gap between T_{alloc} and $T_{preempt}$ allows for more fluctuation in actual utilization, and hence leads to lower preemptions. As noticed earlier, jobs with greater variability have more preemptions.

C. Results from a realistic deployment

To understand the benefits of UBIS in real cluster environments, we deployed our FairScheduler implementation on a 20-node cluster. Table II captures the resources available on each node and the amount of resources allotted for use by YARN containers. Each node had 32 physical cores, of which we allot 28 for use by YARN containers. YARN has the notion of virtual cores (vcores) to allow allocating a fraction of a physical core to a container and realize the notion of a homogeneous CPU unit in a potentially heterogenous cluster. For our experiments, we split each physical core to 2 virtual

Tab. II: Per-node resources on the 20 node deployment

Resource	Node capacity	Yarn
Memory	256 GB	130 GB
CPU	32 cores	28 cores as 56 vcores
Disk	12 disks	12 disks
Network	10G	10G

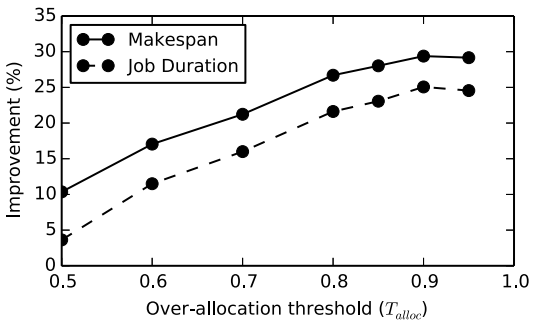


Fig. 6: Percentage improvement in (1) makespan for job-mix and (2) aggregate job duration for different values of T_{alloc}.

cores leading to 56 vcores per node. While we have ample memory (256 GB) on each node, we allocate only 130 GB to YARN, since none of our containers need more than 2 GB. YARN has access to all 12 disks available on each node. Each task is assigned one working directory for temporary files, the most notable being intermediate shuffle data.

The cluster has 1 master node and 19 worker nodes leading to aggregate YARN capacity of 1064 vcores and 2.41 TB of memory.

1) Representative workload

First, we run a workload that represents a typical big-data workload as observed in production deployments. The workload consists of three pipelines submitted simultaneously and run in parallel:

1) A TPC-DS Hive query (query-35) that operates on a 3TB dataset. TPC-DS [16] is an industry-standard benchmark for decision support systems including big-data systems.
2) An ETL (extract-transform-load) pipeline captured by the teragen, terasort and tervalidate jobs. Teragen generates a terabyte of random data representing ingestion of data into a Hadoop cluster (extract). Terasort sorts the data generated by teragen (transform). Teravalidate validates whether the output of Terasort is indeed sorted (load).
3) A CPU intensive job - Wordcount - that counts the occurrences of words in a 1 TB of randomly generated input data. Wordcount represents the class of jobs that compute statistics on a large corpus of data; examples include clickstream counting or log analysis.

These jobs also differ in their I/O access patterns: the proportion of input data read from distributed storage, intermediate data written to local disks, and output data written back to distributed storage [15]. For our evaluation runs, we set the task resource requirements to the peak usage observed in our trial runs with different CPU and memory settings.

Figures 6, 7, and 8 capture the results of running these representative workloads. We focus our analysis on metrics directly impacting the end-user — the makespan of the entire workload (duration between first job's submission and last job's completion), individual job and task durations. For these runs, we vary T_{alloc} and fix $T_{preempt}$ at 0.95 for memory and 0.99 for CPU.

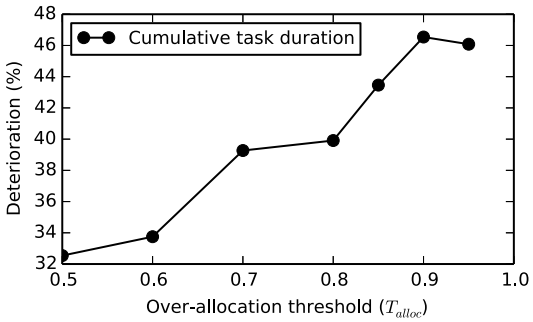


Fig. 7: Percentage deterioration in cumulative task runtimes for different values of T_{alloc}.

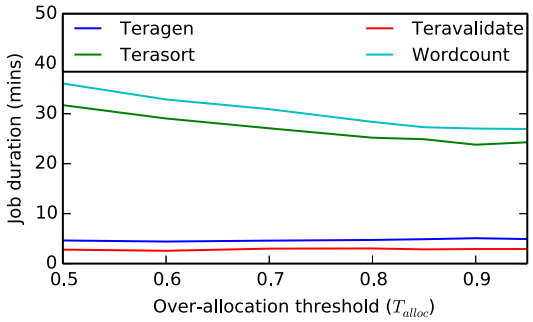


Fig. 8: Individual job durations for different values of T_{alloc}.

Figure 6 plots the improvement in makespan of the entire workload and aggregate job duration (computed as sum of all job durations) against T_{alloc}. The percentage improvement is computed as $(100 \times (baseline - actual)/baseline)$, where $baseline$ corresponds to the traditional (non-UBIS) implementation. The peak improvement is 30% for makespan and 25% for aggregate job duration. Overall, improvement in both metrics increases with T_{alloc} from 0.5 to 0.9 and then decreases from 0.9 to 0.95. As T_{alloc} increases, we see that the percentage of opportunistic containers increases improving cluster utilization and the amount of parallelism. Task contention and number of preemptions also increase steadily with T_{alloc} as the number of opportunistic containers increases leading to task and job slowdowns, respectively. The net improvement in makespan or aggregate job duration is essentially the positive difference between higher utilization (from increased parallelism) and slow-down (due to contention and preemptions). For this workload and set of cluster resources, a value of 0.9 is optimal for T_{alloc}. Note that the dip at 0.95 is marginal, and not as drastic as one would expect. We believe this is because of two reasons: (i) the percentage of opportunistic containers tapers off after a point as discussed in §V-B, and (ii) our implementation is less aggressive with opportunistic allocation at higher values of T_{alloc} as it assumes the containers allocated in the current heartbeat need all allocated resources.

Figure 7 plots the deterioration in aggregate task duration (sum of duration of all tasks in the workload) owing to increased contention discussed earlier. We believe this contention stems from the lack of isolation, especially for disk accesses. Deterioration is computed similar to the improvement $(100 \times |baseline - actual|/baseline)$. As expected, the aggregate task duration steadily increases with T_{alloc}. While we anticipate a strict linear correspondence, we believe the slight deviations from linearity are due to the non-determinism in actual task start times and the duration of overlap.

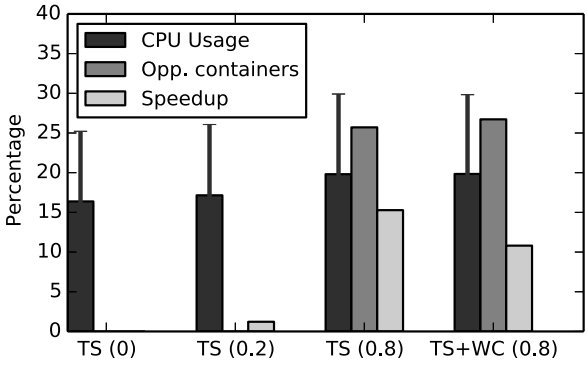


Fig. 9: Scheduler robustness: Percentages of CPU usage, opportunistic containers and speedup for different workload configurations.

The deterioration in task duration, however, does not affect job durations much, due to increased concurrency in the system. Figure 8 plots the actual job duration (in minutes) for the Tera-suite and Wordcount jobs against T_{alloc}. For larger jobs with several waves of map tasks (Terasort and Wordcount), the job duration decreases as T_{alloc} increases before tapering off past 0.9. For smaller jobs with fewer map tasks that all fit in a single wave, the job duration does not improve. At higher values of T_{alloc}, the increased task contention leads to a slight increase in job duration. Better isolation of tasks, including support for disk and network isolation, would help avoid the effect of over-subscription on these smaller jobs.

The optimal values for T_{alloc} and $T_{preempt}$ depend on the cluster and the workload running on it. One can pick $T_{preempt}$ based on the amount of contention the nodes can tolerate. A high value like 0.99 might work well for a malleable resource like CPU as it allows short bursts in usage. For a non-malleable resource like memory, leaving some headroom will allow the worker agent to take corrective action; 0.95 worked fine in our experiments. A good value for T_{alloc} depends more on the workload — workloads with steady usage could use a higher value. Based on our experiments presented in this section, we anticipate a value in the range of 0.75–0.9 to work well for most clusters. Picking the exact optimal value is more involved: trial-and-error is straight-forward and might work well for clusters whose workload characteristics do not change much over time. Gradient descent algorithms could help arrive at the optimal value sooner by restricting the values sampled. One could also use a different value on different nodes and pick the value that corresponds to the best utilization/ preemption tradeoff.

2) Robustness

Having established the gains in throughput for a representative workload, we verify the robustness of UBIS. Specifically, we verify that: (i) UBIS adds minimal overhead on an already well-utilized cluster; and (ii) running small jobs with short tasks opportunistically while running a large job does not affect the job duration of the large job. However, as explained in §II, it is hard to fully utilize a cluster without oversubscription, given the difference in peak and mean usage of tasks. Instead, we set T_{alloc} to the mean node utilization; since the node utilization is already at T_{alloc}, the scheduler should not allocate any opportunistic containers. Figure 9 plots the CPU usage, number of opportunistic containers, and speedup for different workload configurations. TS stands for Terasort on 2 TB data, WC for Wordcount on 100 MB data, and T_{alloc} is in parentheses.

First, we run a large Terasort job that sorts 2 TB of data on the same 20 node cluster without opportunistic scheduling. We notice that the CPU is more contended than memory and focus our analysis on CPU usage. This Terasort run (identified as TS (0) in the figure) forms the base case and is used for speedup calculations of other configurations. For the base case, mean CPU usage is 16.38% with a standard deviation of 9%.

Next, we run the same Terasort job with T_{alloc} set to 0.2. Since the node utilization is already close to this value, we see no opportunistic containers allocated. We see minor increase in CPU utilization (mean of 17.15%) and a corresponding speedup of 1.22%. We attribute this minor improvement to the non-deterministic nature of task execution.

When we increase T_{alloc} to 0.8, as expected, we see opportunistic allocations (25% of total tasks) leading to noticeable improvements in CPU utilization and speedup (15.28%). Now, along with this large Terasort job, we submit a small Wordcount job that counts the number of occurrences of the words in a 100 MB corpus of data. We notice a slight increase in the number of opportunistic containers, with negligible change in CPU usage. These opportunistic containers are allocated to both Terasort and Wordcount jobs. The Terasort job still sees a considerable speedup of 10.8%.

VI. Related Work

Resource scheduling for distributed computing has been an area of active research for a long time, from Condor [19] for HiPC workloads to MapReduce [6] and Dryad [13] for data-centric workloads. For brevity, we limit our discussion to the most relevant work aimed at improving cluster utilizaiton.

Most modern schedulers — Apollo [4], Borg [21], Corona [3], Mesos [12], Omega [18], Quincy [14], YARN [20] — adopt fine grained resource scheduling, where the scheduler allocates resources based on per-task resource estimates; this avoids the internal fragmentation in fixed-size slot based approaches. However, as illustrated in §II, users are typically not equipped to estimate individual task resource requirements.

Instead of resource estimates, Quasar [7] asks users for performance-oriented requirements (latency, runtime) for the workloads. Quasar then uses fast classification techniques, based on some profiling information and data from previous workloads, to determine workload characteristics including the impact of interference. Quasar uses this information to identify the best workloads to run together and resource allocations.

Other recent efforts focus on improving existing cluster schedulers. Yaq [17] proposes proactively queuing containers on worker nodes to be executed as soon as resources become available, instead of reactively scheduling on node heartbeats. Tetris [11] employs multi-dimensional bin-packing to run containers with resource requirements complementary to the current usage on the node to reduce contention. UBIS complements both these approaches and we believe integrating with these approaches would lead to further improvements in utilization and job throughput.

Apollo [4] and Borg [21] both propose a hierarchy of tasks, where some tasks run with better SLAs than others, and are closest to our work. Apollo adopts token-based scheduling and proposes using opportunistic allocations similar to our approach, but does not base its allocation on actual utilizaiton. Instead, it relies on application schedulers optimistically requesting opportunistic containers. Apollo does not guarantee fairness either, but employs probabilistic resource fairness to limit unfairness.

Borg [21] employs priority based admission control and scheduling; the jobs can be broadly classified into *prod* (high priority monitoring and long-running production services/jobs) and *non-prod* (lower priority batch and best-effort) jobs. Borg *reclaims* resources from running tasks and could allocate these resources to the lower priority *non-prod* tasks; the latter can get preempted if the machine runs out of resources. While these non-prod tasks are similar to our opportunistic containers, there are certain differences. Borg differentiates between prod and non-prod tasks, but does not track tasks that use reclaimed resources separately. By tracking opportunistic containers separately, UBIS allows jobs to decide what to run in these containers; for instance, a MapReduce job could choose to run only maps in opportunistic containers. Borg does not allow individual jobs to choose the number of opportunistic containers. Borg does not support fairsharing either, which is critical to enterprise users.

VII. Limitations and Future Work

In the process of evaluating our approach, we have identified other potential improvements to our approach and implementation.

Support for I/O resources. Our current implementation in YARN oversubscribes CPU and memory, without considering I/O resources. As outlined in Tetris [11], this can lead to untenable oversubscription of resources that are not considered. Actively scheduling and isolating I/O resources can help reduce contention on these resources and limit the deterioration in task runtimes.

Faster memory monitoring. Current memory monitoring in YARN, through aggregation of procfs metrics, does not scale to hundreds of containers we observed. This can be improved by using advanced operating system libraries like cgroups in

Linux.

Cross-node container promotion. UBIS avoids promoting containers across nodes to avoid losing work. If tasks preserve their work, say through check-pointing, cross-node promotions could help in satisfying constraint C4 for all applications without them having to opt-out of opportunistic containers.

Picking the optimal over-allocation threshold. As discussed earlier in §V-C1, cluster administrators are expected to arrive at the optimal value for T_{alloc} through trial-and-error. Automating this process with the ability to adapt to changing workload patterns over time would be very useful.

Suitability of individual jobs. Jobs differ in their suitability to opportunistic containers. For instance, jobs with short tasks are likely more amenable — (1) they do not run long enough to cause sustained contention that requires preemptions, (2) any savings in allocation latency is significant given their short runtime. Streaming jobs or long-running services, on the other hand, are likely less suited for the same reasons. Providing a way to tune opportunistic allocations on a per-job basis and leveraging past history to determine the amenability of jobs are promising avenues for future work.

Queuing opportunistic tasks. Cluster schedulers adopt a reactive approach to scheduling: they schedule containers when the node reports the availability of resources. Queuing opportunistic containers, similar to [17], allows the nodes to execute tasks as soon as they finish one, thereby improving the utilization further.

VIII. Conclusion

In current distributed environments, significant amount of resources are wasted due to tasks not utilizing all allocated resources. We attribute this to cluster schedulers that consider only user-specified resource requirements and not actual container utilization. Improving utilization while honoring fairness, without adversely affecting application SLAs is hard. We present UBIS, a utilization-aware approach to scheduling, that allocates un-utilized resources to pending tasks and gracefully handles resource pressure on the worker nodes. UBIS guarantees fairness by sharing both un-allocated and un-utilized resources as per the specified scheduler policies, and applications can opt out of opportunistic allocations to ensure their containers are not preempted under resource pressure. UBIS implementation in YARN shows up to 30% improvement in workload makespan.

Acknowledgements

We thank the Apache YARN community for discussions and suggestions, particularly Yahoo! for sharing their cluster statistics and feedback from their prior attempts at oversubscription. This work is supported by NSF grants CSR 1422338 and CCF 1533795.

References

[1] Apache Hadoop YARN: FairScheduler. https://s.apache.org/fair-scheduler.

[2] Apache Hive. http://hive.apache.org.

[3] Corona. https://s.apache.org/corona.

[4] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, 2014.

[5] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes. Long-term SLOs for reclaimed cloud computing resources. In *SoCC*, 2014.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[7] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *ASPLOS*, 2014.

[8] A. Desai, K. Rajan, and K. Vaswani. Critical path based performance models for distributed queries. In *Microsoft Technical Report*, 2012.

[9] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[10] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *EuroSys*, 2013.

[11] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[12] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[14] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[15] K. Kambatla and Y. Chen. The truth about mapreduce performance on ssds. In *LISA*, 2014.

[16] R. O. Nambiar and M. Poess. The making of tpc-ds. In *VLDB*, 2006.

[17] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *Microsoft Research Tech Report*, 2016.

[18] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[19] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. In *CPE*, 2005.

[20] 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 *SoCC*, 2013.

[21] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *EuroSys*, page 18. ACM, 2015.