Efficient Queue Management for Cluster Scheduling

Jeff Rasley[◇†] Konstantinos Karanasos[†] Srikanth Kandula[†]
Rodrigo Fonseca[◇] Milan Vojnovic[†] Sriram Rao[†]

[†]*Microsoft* [◇]*Brown University*

Abstract

Job scheduling in Big Data clusters is crucial both for cluster operators' return on investment and for overall user experience. In this context, we observe several anomalies in how modern cluster schedulers manage queues, and argue that maintaining queues of tasks at worker nodes has significant benefits. On one hand, centralized approaches do not use worker-side queues. Given the inherent feedback delays that these systems incur, they achieve suboptimal cluster utilization, particularly for workloads dominated by short tasks. On the other hand, distributed schedulers typically do employ worker-side queuing, and achieve higher cluster utilization. However, they fail to place tasks at the best possible machine, since they lack cluster-wide information, leading to worse job completion time, especially for heterogeneous workloads. To the best of our knowledge, this is the first work to provide principled solutions to the above problems by introducing queue management techniques, such as appropriate queue sizing, prioritization of task execution via queue reordering, starvation freedom, and careful placement of tasks to queues. We instantiate our techniques by extending both a centralized (YARN) and a distributed (Mercury) scheduler, and evaluate their performance on a wide variety of synthetic and production workloads derived from Microsoft clusters. Our centralized implementation, *Yaq-c*, achieves $1.7\times$ improvement on median job completion time compared to YARN, and our distributed one, *Yaq-d*, achieves $9.3\times$ improvement over an implementation of Sparrow's batch sampling on Mercury.

1. Introduction

Data-parallel frameworks [7, 27, 32] and scale-out commodity clusters are being increasingly used to store and extract value from data. While some enterprises have large clusters, many others use public cloud providers. Such clusters run a wide variety of applications including batch data analytics jobs, machine learning jobs and interactive queries. To reduce operational costs, and, therefore, improve return on investment, there is a trend toward consolidating diverse workloads onto shared clusters. However, doing so places considerable strain in the cluster scheduler, which has to deal with vastly heterogeneous jobs, while maintaining high cluster utilization, fast and predictable job completion times, and offering expressive sharing policies among users.

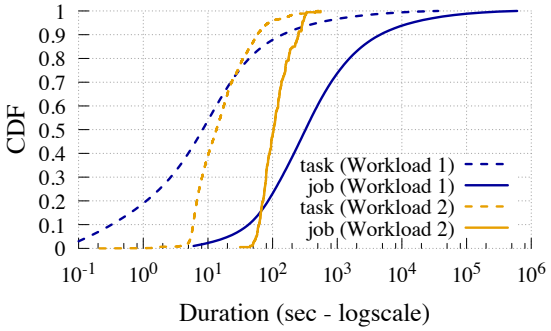

Figure 1. Job and task durations for production workloads.

To showcase the job heterogeneity in production clusters, we provide task and job durations for two production workloads of Microsoft (see Figure 1).[1] Task durations vary from a few milliseconds to tens of thousands of seconds. Moreover, a significant fraction of tasks are short-lived ($\sim$50% last less than 10s), which illustrates a generally observed shift towards smaller tasks [20–22].

Cluster schedulers such as YARN [27] and Borg [28] have a logically centralized service, often called the resource manager (RM), which serves as a matchmaker between the resource needs of various jobs and the available resources on worker machines. Typically, machines exchange heartbeat messages with the RM once every few seconds,[2] and

EuroSys '16, April 18 - 21, 2016, London, United Kingdom

ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2901318.2901354

[1] Durations only for successful (non-failed) tasks are included in the figure.

[2] YARN clusters with $\sim$4K nodes use a heartbeat interval of 3sec [27]; the Borg RM polls each machine every few secs with the 95th percentile being below 10sec in a $\sim$10K-node cluster [28].

are initiated either by worker machines (in YARN) or by the RM (in Borg). Through heartbeats, worker machines report resource availability to the RM, which in turn determines the allocation of tasks to machines. This design has two main problems: first, the RM is in the critical path of all scheduling decisions; second, whenever a task finishes, resources can remain fallow between heartbeats. These aspects slow down job completion: a job with a handful of short tasks can take tens of seconds to finish. Worse, they affect cluster utilization especially when tasks are short-lived. Table 1 shows the average cluster utilization (i.e., the percentage of occupied slots) with tasks of different durations for an 80-node YARN cluster. The label X sec denotes a synthetic workload wherein every task lasts X seconds. The label Mixed-5-50 is an even mix of 5 and 50 sec tasks. Workload 1 is the production workload shown in Figure 1. We see that as task durations get shorter, cluster utilization drastically degrades, and can be as low as 61%.

A few schedulers avoid logical centralization. Apollo [8], Sparrow [22] and others allow job managers to independently decide where to execute their tasks, either to improve scalability (in terms of cluster size or scheduling decisions rate) or to reduce allocation latency. The above problem with short-lived tasks becomes less prevalent, because tasks can be *pushed* onto queues at worker machines by each job manager. However, these schedulers are vulnerable to other problems: (a) each job manager achieves a *local* optimum allocation, but coordination across various job managers to achieve *globally* optimal allocations is not possible;[3] (b) worse, the distributed schedulers do not always pick appropriate machines since they fail to account for the pending work in each queue; (c) the assignments are vulnerable to head-of-line blocking when tasks have heterogeneous resource demands and durations. These aspects affect job completion times, leading to increased tail latency and unpredictability in job run times.

To illustrate these aspects, Figure 2 presents a CDF of job completion times for Workload 2 with YARN and an implementation of Sparrow's batch sampling on Mercury [20, 22]. We see that the latter improves some very short jobs, but has a long tail of jobs that exhibit longer completion times. As we will see later, this happens because batch sampling fails to make globally optimal task placement decisions, and because FIFO queues at worker nodes suffer from head-of-line blocking. Moreover, to address the utilization problems mentioned above for centralized schedulers, we extended YARN by allowing tasks to be queued at each node, thus masking task allocation delays. In this case, the RM assigns tasks to node queues in a way that is similar to how it already assigns tasks to nodes. The resulting job completion times are depicted in the "YARN+Q" line of Figure 2. We see that

[3] For example, when scheduling a task of job_1 with equal preference for machines $\{m_1, m_2\}$ and a task of job_2 that will run much faster at m_1, it is not possible to guarantee that job_2's task will always run at m_1.

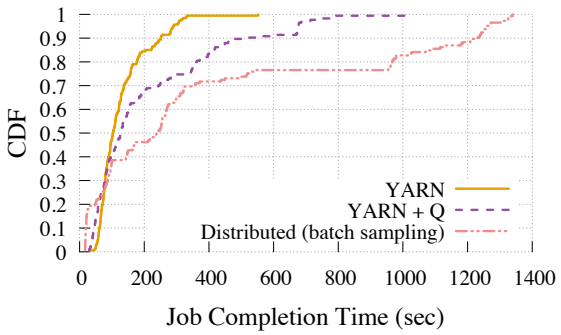

Figure 2. Job completion times for production Workload 2 using different scheduling types.

5 sec	10 sec	50 sec	Mixed-5-50	Workload 1
60.59%	78.35%	92.38%	78.54%	83.38%

Table 1. Average YARN cluster slot utilization for workloads with varying task durations.

naively offering FIFO queues at worker nodes in YARN can be worse than not having queues at all. As will be shown later, this is due to similar head-of-line issues, as well as the potentially poor early binding of tasks to machines.

In this paper, we provide more principled solutions towards using task queues well in the context of cluster schedulers, be they centralized or distributed. Our contributions include:

- We present a centralized (Yaq-c) and a distributed (Yaq-d) cluster scheduling design, both of which support queuing at worker nodes and can accommodate our queue management techniques, without losing the attractive features of existing designs (Section 2).
- We show that naively adding queues at worker nodes is unlikely to work well. Rather, we offer guidance on how to bound the length of queues: using short queues can lead to lulls and thus to lower cluster utilization, whereas using long ones increases queuing delays and encourages sub-optimal early binding of tasks. We also study how to place tasks to worker machines when doing so involves picking a run-slot, a queue-slot, or waiting. We finally introduce task prioritization techniques that are novel to worker-side queues in cluster scheduling, and which are crucial for reducing head-of-line blocking and improving job completion times (Section 3).
- We study how the per-queue scheduling techniques above can be combined well with cluster-wide policies, such as global job prioritization (in centralized designs) and cluster sharing policies (Section 4).
- We implemented both Yaq-c and Yaq-d (Section 5), and deployed them on an 80-node cluster. Our experimental results using synthetic and production workloads (derived

	Scheduling type	Type of queuing		Queue management		
Scheduling framework	Centralized/Distributed	Global queue	Queues at nodes	Task placement	Queue sizing	Queue reordering
YARN [27]	✓/-	✓		∗		
Borg [28]	✓/-	✓		∗		
Sparrow [22]	-/✓		✓	✓		
Apollo [8]	-/✓		✓	✓		
Mercury [20]	✓/✓	✓	✓	✓		
Yaq-c	✓/-	✓	✓	✓	✓	✓
Yaq-d	-/✓		✓	✓	✓	✓

Table 2. Overview of queuing capabilities of existing scheduling frameworks compared to Yaq (∗ indicates that the system performs placement of tasks to nodes but not to queues).

from Microsoft clusters) show that Yaq-c improves median job completion time by $1.7\times$ over stock YARN. Relative to an implementation of Sparrow's batch sampling [22] on YARN and of Mercury [20], Yaq-d improves median job completion time by $9.3\times$ and $3.9\times$, respectively.

We plan to release both Yaq-c and Yaq-d by contributing them to Apache YARN. The support for queuing of tasks at YARN's worker nodes is already available at Apache JIRA YARN-2883 [30]. The rest of the work will be released in related JIRAs.

2. Design

In this section, we describe the design of our two cluster scheduler variations, *Yaq-c* and *Yaq-d*, upon which we implement and evaluate our queue management techniques. Yaq-c extends the centralized scheduler in YARN [27] by adding task queues at worker nodes. Yaq-d, on the other hand, is a distributed scheduler that extends our Mercury scheduler [20, 29]. After laying out the requirements for our scheduler (Section 2.1), we first give an overview of our queuing techniques and compare Yaq's capabilities with those of existing scheduling frameworks (Section 2.2). Then, we present the basic components of our system design (Section 2.3), and detail the specifics of our design for both Yaq-c and Yaq-d (Section 2.4 and Section 2.5).

2.1 Requirements

Resource managers for large shared clusters need to meet various, often conflicting, requirements. Based on conversations with cluster operators and users, we distill the following set of requirements for our system.

Heterogeneous jobs: Due to workload consolidation, production clusters have to simultaneously support different types of jobs and services (e.g., production jobs, best-effort jobs). Hence, tasks have highly variable durations and resource needs (e.g., batch jobs, ML, MPI, etc.).

High cluster utilization: Since cluster operators seek to maximize return on investment, the scheduler should optimally use the cluster resources to achieve high cluster utilization. The expectation is that higher cluster utilization leads in turn to higher task and job throughput.

Fast (and predictable) job completion time: Cluster users desire that their jobs exit the system quickly, perhaps as close as possible to the jobs' ideal computational time. Furthermore, predictable completion times can substantially help with planning.

Sharing policies: Since the cluster is shared amongst multiple users, operators require support for sharing policies based on fairness and/or capacity constraints.

2.2 Task Queuing Overview

As we will describe in Sections 3 and 4, the introduction of local queues in Yaq-c, and the management of the different queues in both Yaq-c and Yaq-d are our key contributions. It is thus useful to contrast our designs with existing systems.

In Table 2, we outline the type of queuing that existing systems enable (global queuing and/or local at the nodes), as well as the queue management capabilities they support compared to Yaq. Due to their inherent design, distributed and hybrid schedulers (such as Sparrow, Apollo, Mercury) support queuing at the nodes, but not global job queuing. On the other hand, to the best of our knowledge, no existing centralized system supports queuing at worker nodes. This is an interesting point in the design space that we explore in Yaq-c. Further, although all systems with queues at worker nodes need to implement a task placement policy, none of them implement additional queue management techniques, such as task prioritization through queue reordering, and queue sizing. Hence, we explore such techniques in Yaq-c and Yaq-d.

2.3 Basic System Components

The general system architecture, depicted in Figure 3 (for Yaq-c) and Figure 4 (for Yaq-d), consists of the following main components:

Node Manager (NM) is a service running at each of the cluster's worker nodes, and is responsible for task execution at that node. Each NM comprises *running tasks* and *queued tasks* (as shown in Figures 3 and 4). The former is a list with the tasks that are currently being executed, thus occupying actual resources at the node. The latter is a queue with the tasks that are waiting on the resources held

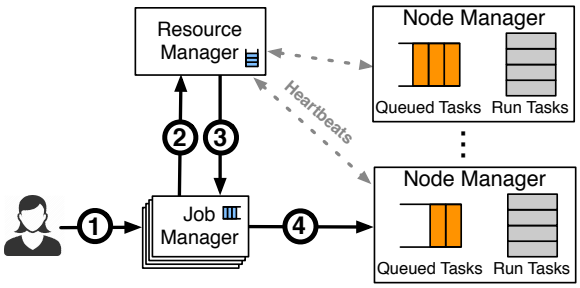

Figure 3. System architecture for centralized scheduling (Yaq-c).

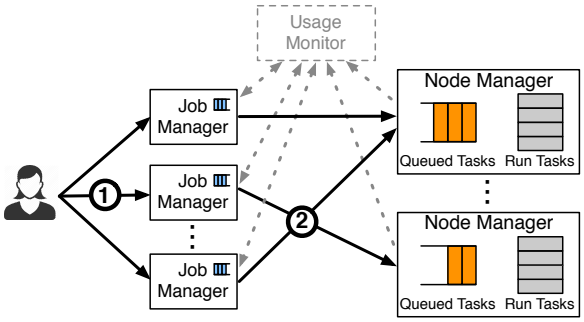

Figure 4. System architecture for distributed scheduling (Yaq-d).

by the currently running tasks and are thus not occupying actual resources. A task is queued only if the NM cannot start its execution due to insufficient resources.

Resource Manager (RM) is the component that manages the cluster resources in centralized scheduling settings (thus appears only in Yaq-c). The NMs periodically inform the RM about their status through a heartbeat mechanism. Based on the available cluster resources and taking into account various scheduling constraints (e.g., data locality, resource interference, fairness/capacity) and a queue placement policy (to determine where tasks will be queued, if needed), it assigns resources to tasks for execution.

Usage Monitor (UM) is a centralized component to which the NMs periodically report their status. It is used in distributed scheduling frameworks as a form of loose coordination to perform more educated scheduling decisions. Although this component is not necessary [22], a form of a UM has been used in existing distributed schedulers [8, 20], and is also used in Yaq-d.

Job Manager (JM) is a per-job orchestrator (one JM gets instantiated for each submitted job). In centralized settings, it negotiates with the RM framework for cluster resources. Once it receives resources, it dispatches tasks for execution to the associated nodes. In distributed settings, where there is no central RM, it also acts as a scheduler, immediately dispatching tasks to nodes.

2.4 Centralized Scheduling With Queues (Yaq-c)

Our system architecture for centralized scheduling is depicted in Figure 3. As shown in the figure, a job's lifecycle comprises the following steps. First, as soon as a client submits a new job to the cluster, the JM for this job gets initialized (step 1). The tasks of the job get added to the queue that is maintained locally in the JM. Then, the JM petitions the RM for cluster resources based on the resource needs of the job's tasks (step 2). The RM chooses where to place the tasks based on some policy (such as resource availability, status of queues at the NMs, data locality, etc.), and then notifies the JM (step 3). Subsequently, the JM dispatches the tasks for execution at the specified nodes (step 4). A task will start execution whenever it is allocated resources by the NM, and until that moment it is waiting at the NM's queue. The job's lifecycle terminates when all of its tasks complete execution.

Note that the RM performs job admission control, based on the available resources and other constraints (e.g., cluster sharing policies). Thus, when a job is submitted, it waits at a global queue in the RM (shown in blue in the figure), until it gets admitted for execution.

2.5 Distributed Scheduling With Queues (Yaq-d)

Our system architecture for distributed scheduling is shown in Figure 4. When a client submits a new job, the corresponding JM gets instantiated (step 1 in the figure). The JM, who is now acting as the task scheduler for that job, uses a scheduling policy to select the node to which each of the job's task will be dispatched. The scheduling policy relies also on information that becomes available from the UM, such as the queue status of a node. The JM then places the tasks to the specified nodes for execution (step 2). Similar to the centralized case, if resources in a node are available, task execution starts immediately. Otherwise, the task waits in the queue until resources become available.

Our design also enables restricting the number of concurrently executing or queued tasks per JM. We defer the details to Section 5.

3. Queue Management at Worker Nodes

In the design outlined so far, queues at worker nodes are of particular importance since they determine when a task bound to a node starts execution. This is the case with either architecture, centralized or distributed. However, as explained in Section 1, simply maintaining a queue of tasks waiting for execution at worker nodes does not directly translate to benefits in job completion time, especially in the presence of heterogeneous jobs.

To this end, our main focus in this work is on efficiently managing local node queues. Our queue management includes the following techniques: (1) determine the queue length (Section 3.1); (2) decide the node to which each task

will be placed for queuing (Section 3.2); (3) prioritize the task execution by reordering the queue (Section 3.3). We discuss cluster-wide queue management policies in Section 4.

Note that placing tasks to queues is required whenever the actual cluster resources are not sufficient to accommodate the jobs that are submitted in the cluster. Thus, our techniques become essential under high cluster load. In cases of lower cluster load, when no worker-side queuing is needed, Yaq-c behaves like YARN and Yaq-d like Mercury.

To simplify our analysis, in this section we consider *slots* of resources consisting of memory and CPU, as done in YARN too. Whenever applicable, we discuss how our techniques can be extended to support multi-dimensional resources.

Task duration estimates Part of our work relies on estimates of task durations, based on the observation that in our production clusters at Microsoft, more than 60% of the jobs are recurring. For such jobs, we assume an initial estimate of task durations based on previous executions. As we show in our experiments, Yaq performs well even with only rough estimates (such as the average duration of a map or reduce stage).[4] In the absence of such estimates, we assume a default task duration and have extended the JM to observe actual task durations at runtime and refine the initial estimate as the execution of the job proceeds.

3.1 Bounding Queue Lengths

Determining the length of queues at worker nodes is crucial: queues that are too short lead to lower cluster utilization, as resources may remain idle between allocations; queues that are too long may incur excessive queuing delays. We discuss two mechanisms for bounding queue lengths: length-based bounding (Section 3.1.1) and delay-based bounding (Section 3.1.2).

3.1.1 Length-Based Queue Bounding

In length-based queue bounding, all nodes have a predefined queue length b, and the RM can place up to b tasks at the queue of each node. We now focus on how to determine the value of b. We first consider the case when all tasks have the same duration, and then turn to the more general case.

Note that we base our analysis on the centralized design, where task placement is heartbeat-driven. We defer the analysis for the distributed case for future work, but expect the findings to be largely similar.

Fixed task duration We assume that all tasks have the same duration $1/\mu$ (where μ is the task processing rate), and calculate the minimum queue length that would guarantee a desired cluster utilization. Let r be the maximum number of tasks that can run concurrently at a node (based on its resources and the minimum resource demand of a task), and

[4] Note that more sophisticated models for estimating task durations can be employed. We purposely opted for a simpler approach here, to assess our system's behavior even with inaccurate estimates.

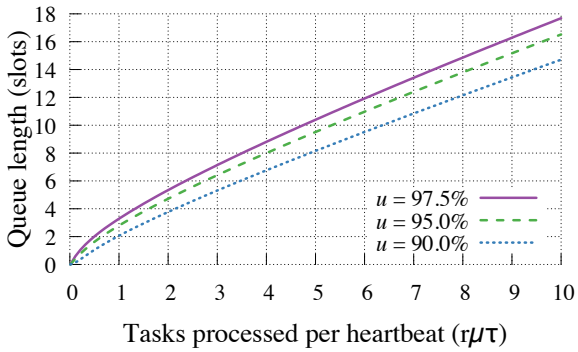

Figure 5. Queue length required to achieve cluster utilization $u{=}1{-}\delta$, given the number of tasks that can be processed by a node per heartbeat interval ($r\mu\tau$).

τ the heartbeat interval. Then the maximum task processing rate at the node is $r\mu$. Given r running tasks and b queued tasks, a node will remain fully utilized when: $r + b \geq r\mu\tau$ or $b \geq r(\mu\tau - 1)$.

Interestingly, the above reasoning is similar to the bandwidth-delay product for TCP flows, where the goal is to have enough packets in flight to keep the link fully utilized. In cluster scheduling, tasks can be seen as analogous to packets, node processing rate to the link capacity, and heartbeat interval to RTT.

Exponentially-distributed task duration We consider an arbitrary node that has r run slots and a queue of length b slots. We want to determine the value of parameter b such that node utilization is at least $1 - \delta$ for given parameter $\delta \in (0, 1]$. Here we provide the main results of our analysis; more details along with our proofs can be found in Appendix A. We note that node utilization is at least as large as the fraction of heartbeat intervals in which all run slots are always busy. It thus suffices to configure the queue length so that the latter quantity is at least of value $1 - \delta$.

We admit the following assumptions. Whenever the node completes processing a task, we assume that it starts processing one of the tasks from the queue taken uniformly at random, if there are any in the queue. We assume that task processing times are independent and identically distributed according to an exponential distribution with mean $1/\mu$. This assumption enables us to characterize the node utilization by leveraging the memory-less property of the exponential distribution.

Proposition 1. *At least a $1{-}\delta$ fraction of heartbeat intervals will have all run slots always busy, if the queue length b is at least as large as the smallest integer that satisfies*

$$r\mu\tau\left(1+\frac{b+1}{r\mu\tau}\left(\log\left(\frac{b+1}{r\mu\tau}\right)-1\right)\right)\geq\log\left(\frac{1}{\delta}\right). \quad (1)$$

We now discuss the asserted sufficient condition. If the task processing times were deterministic assuming a com-

mon value $1/\mu$ and the length of the heartbeat interval is a multiple of $1/\mu$, then for 100% utilization it is necessary and sufficient to set the queue length such that $b + r = r\mu\tau$. This yields the queue length that is linear in $r\mu\tau$, for any fixed value of the run slots r. The sufficient condition in (1) requires a larger queue length than $r\mu\tau$ for small values of $r\mu\tau$. It can be shown that the sufficient condition (1) requires the queue length that is at least $r\mu\tau + \sqrt{\log(1/\delta)}\sqrt{r\mu\tau}$, for large $r\mu\tau$. For numerical examples, see Figure 5. Specifically, given a heartbeat interval $\tau = 3$ sec, an average task duration $1/\mu$ of 10 sec, $r = 10$ tasks allowed to be executed at a node at the same time, and a target utilization of 95%, a queue of $b = 6$ slots is required. Likewise, for an average task duration of 30 sec, the queue size should be ≥ 3 slots. These values for b are also validated by our experiments (Section 6) on the production Workload 2 of Figure 1.

3.1.2 Delay-Based Queue Bounding

Maintaining queues of the same fixed length across all nodes does not work well with heterogeneous tasks. When short tasks happen to be present in a node, this may lead to underutilization of its resources, whereas when tasks are longer, significant delays may incur. Hence, when task durations are available, we use a *delay-based* strategy. This strategy relies on the estimated queue wait time that gets reported by each node at regular intervals, as we explain in Section 3.2 (Algorithm 2). In particular, we specify the maximum time $\mathrm{W}T_{max}$ that a task is allowed to wait in a queue. When we are about to place a task t at the queue of node n (see Section 3.2), we first check the last estimated queue wait time $\mathrm{W}T_n$ reported by n. Only if $\mathrm{W}T_n < \mathrm{W}T_{max}$ is t queued at that node. Upon queuing, the RM uses a simple formula to update $\mathrm{W}T_n$ taking into account t's task duration estimate, until a fresh value for $\mathrm{W}T_n$ is received from n. Using this method, the number of tasks that get queued to each node gets dynamically adapted, based on the current load of the node and the tasks that are currently running and queued.

Note that this technique can be directly applied in both our centralized and distributed designs.

3.2 Placement of Tasks to Nodes

Given a job consisting of a set of tasks, the scheduler has to determine the nodes to which those tasks will be placed. We now present the algorithm that Yaq uses for task placement. We also introduce the algorithm that we use to estimate the time a task has to wait when placed in a node's queue before starting its execution. This algorithm is crucial for high quality task placement decisions.

As explained in Section 2, we assume that there is a central component to which each node periodically publishes information about its resource and queue status. This component is the RM in Yaq-c (see Figure 3), and the UM in Yaq-d (see Figure 4).

Algorithm 1: Placement of task to node

Input : t: task to be placed; Rf_{min}: min free cluster resources percentage before starting to queue tasks
Output: node where t will be placed

```
// Avoid queuing when available cluster resources
1 if freeResources / totalResources > Rf_min then
2     return placeTaskNoQueuing(t)
3 else return node n with highest queuingScore(n, t)

// How suitable is node n for placing task t to its queue
4 Function queuingScore(n, t)
      // affScore ∈ (0, 1] based on data locality (or resource
         interference) when placing t on n (higher is better)
5     affScore ← affinityScore(n, t)
      // Compute load of node based on queue length or
         queue wait time (using Algorithm 2)
6     nload ← nodeLoad(n)
7     return affScore × 1/nload
```

Our task placement algorithm is outlined in Algorithm 1. It takes as input a task t and outputs the node n where t should be placed. Yaq preferentially places tasks at nodes that have available resources since such tasks will incur no queuing delays. Thus, we first check if there are such resources (line 1), and if so, we place t to a node with available local resources, taking other parameters such as data locality also into account (line 2). If the cluster is almost fully loaded (as defined by the $\mathrm{R}f_{min}$ parameter given as input), we choose which node's queue to place t (line 3). We use the function `queuingScore(n,t)` to quantify how suitable a node is for executing t. The score of a node comprises two components: node affinity for t and node load. In our current implementation, node affinity takes into account data locality, but this can be extended to also consider resource interference, providing better resource isolation when executing t. The load of the node can be calculated based on one of the following strategies depending on the richness of the information published by each node:

Based on queue length: The simplest information that each node publishes is the size of its queue. This strategy gives higher score to nodes with smaller queue lengths. Note that this can lead to suboptimal placement decisions in case of heterogeneous tasks: a node with two queued tasks of 500 secs each will be chosen over a node with five tasks of 2 secs each.

Based on queue wait time: This strategy assumes that each node publishes information about the estimated time a task will have to wait at a node before starting its execution, as described below. The lower this estimated wait time is, the higher the score of the node. This strategy improves upon the previous one when considering heterogeneous tasks, as we also show experimentally in Section 6.5.2.

Algorithm 2: Estimate queue wait time at node

Input : $runTasks$: running tasks' remaining durations; $queuedTasks$: queued tasks' durations; $freeResources$: free node resources; $freeResources_{min}$: min free node resources before considering a node full

Output: Estimated queue wait time for the next task that will be dispatched to the node

```
1  if freeResources ≥ freeResources_min then
2      return 0
3  waitTime ← 0
4  for qTask in queuedTasks do
5      minTask ← remove_min(runTasks)
6      waitTime ← waitTime + minTask
7      runTasks ← [t − minTask for t in runTasks]
8      runTasks.add(qTask)
9  return waitTime + remove_min(runTasks)
```

Algorithm 3: Compare task priorities

Input : tasks t_1, t_2; comparison strategy `taskCmp`; hard starvation threshold ST; relative starvation threshold ST_r

Output: > 0 if t_1 has higher priority, < 0 if t_2 has higher priority, else 0

```
1  if isStarved(t1) xor isStarved(t2) then
2      if isStarved(t1) then return +1
3      else return −1
4  if !isStarved(t1) and !isStarved(t2) then
5      cmp ← taskCmp(t1, t2)
6      if cmp ≠ 0 then return cmp
7  if isStarved(t1) and isStarved(t2) then
8      cmp ← t2.jobArrivalTime − t1.jobArrivalTime
9      if cmp ≠ 0 then return cmp
10 return t1.queueTime − t2.queueTime
   ---------------------------------------------
11 Function isStarved(ti)
12     return
       ti.queueTime > min(ST, ti.durationEst × STr)
```

Note that Algorithm 1 suggests that we calculate the score of all nodes for placing each task. Clearly this can lead to scalability issues, thus in practice we apply various optimizations (e.g., compute the score of each node only at regular intervals).

Estimating queue wait time at worker nodes Algorithm 2 outlines how each worker node independently estimates the expected queuing delay that a new task will incur if it is placed in its queue. Queue wait time estimates are then periodically sent to the RM (in Yaq-c) or UM (in Yaq-d) to help with task placement. Effectively, the algorithm simulates CPU scheduling. It takes as input the remaining durations of the currently running tasks, and the durations of the queued tasks.[5] If there are available resources, the new task will not have to wait (line 2). Otherwise, we go over the queued tasks and accumulate the time that each of them has to wait before its execution starts (lines 4-8). The first task in the queue will have to wait for the running task with the smallest remaining duration to finish. Then that task gets removed from the running task list (line 5), and its task duration gets added to the accumulated queue wait time (line 6). All remaining running task durations get updated (line 7), the first task in the queue gets added to the list of running tasks (line 8), and the same process repeats for all queued tasks.

Observe that in our estimation, we make the assumption that a queued task can take the slot of any previously running task. One could extend our algorithm to take into account the exact resources required by each task, similar to the queue-wait time matrix of Apollo [8].

Observe that from the moment a task gets placed to a node's queue until the moment its execution starts, better placement choices may become available. This may be due to incorrect information during initial task placement (e.g., wrong queue load estimates) or changing cluster conditions (e.g., resource contention, node failures). Various *corrective actions* can be taken to mitigate this problem, such as dynamic queue rebalancing [20], duplicate execution [2, 8, 9] or work stealing [10]. Since duplicate execution hurts effective cluster utilization, and work stealing makes it hard to account for locality and security constraints in a shared cluster, in Yaq we use queue rebalancing. However, any other technique could be used instead.

[5] These are *estimations* of task durations, as explained in the beginning of Section 3.

3.3 Prioritizing Task Execution

The queue management techniques presented so far are crucial for improving task completion time: they reduce queuing delay (Section 3.1) and properly place tasks to queues (Section 3.2). However, as we also show experimentally in Section 6, they do not improve job completion time on their own most of the time. This is because they execute queued tasks in a FIFO order, without taking into account the characteristics of the tasks and of the jobs they belong to. To this end, we introduce a task prioritization algorithm that reorders queued tasks and can significantly improve job completion times (see Section 6).

Our prioritization algorithm is generic in that any *queue reordering strategy* can be easily plugged in. Moreover, it is *starvation-aware*, guaranteeing that no task will be starved due to the existence of other higher priority tasks. We implemented various reordering strategies, which we present below. Among them, a significant family of strategies are the *job-aware* ones, which consider *all of the tasks in a job* during reordering. In particular, we emphasize on Shortest

Remaining Job First that gave us the best results in our experiments.

Our task prioritization algorithm is outlined in Algorithm 3. It takes as input two tasks, one of the reordering strategies `taskCmp` (among the following, which we detail below: SRJF, LRTF, STF, EJF), as well as a hard and a relative starvation threshold. Tasks are marked as starved, as explained below, using the function `isStarved` (lines 11-12). Starved tasks have higher priority than non-starved ones (lines 1-3). If none of the tasks are starved, we compare them with `taskCmp` (lines 4-6). If both are starved, we give higher priority to the task of the earlier submitted job (lines 7-9). We finally break ties by comparing the time each task has waited in the queue.

Queue reordering strategies: We have implemented and experimented with the following reordering strategies:

Shortest Remaining Job First (SRJF) gives highest priority to the tasks whose jobs have the least remaining work. The remaining work for a job j is a way to quantify how close j is to completion. It is computed using the formula $\Sigma_{t_i \in RT(j)} t_i \cdot td(t_i)$, where $RT(j)$ are the non-completed tasks of j and $td(t_i)$ is the (remaining) task duration of task t_i, based on our task duration estimates. The remaining work gets propagated from the RM (in Yaq-c) or the UM (in Yaq-d) to the nodes through the existing heartbeats.

Least Remaining Tasks First (LRTF) is similar to SRJF, but relies on number of remaining tasks to estimate the remaining work. Although this estimate is not as accurate as the one used by SRJF, it is simpler in that it does not require task duration estimates. The remaining tasks number gets propagated from the JM to the nodes through the existing heartbeats.

Shortest Task First (STF) orders tasks based on increasing expected duration. This strategy is the only one in this list that is not *job-aware*, given that it uses only task information and is agnostic of the status of the job the tasks belong to. However, it can become interesting when coupled with our starvation-aware techniques.

Earliest Job First (EJF) orders tasks based on the arrival time of the job that the tasks belong to. This is essentially FIFO ordering, and is the default strategy in most schedulers. No additional knowledge is required from the jobs. Although EJF performs no reordering, as described above, we use it to discriminate between starved tasks.

Commonalities between our reordering strategies and existing OS and network scheduling strategies are discussed in Section 7. Observe that our current strategies are oblivious to the *job structure* (e.g., whether a task belongs to the map or reduce phase of a M/R job, the structure of a DAG job, etc.). As part of ongoing work, we are evaluating novel strategies that account for job structure which can be used to further prioritize task execution. Moreover, we are currently investigating how, in the presence of multi-dimensional resources, we can momentarily violate a reordering strategy in order to provide better resource packing and thus achieve higher resource utilization.

Starvation-aware queue reordering All of the above strategies except EJF can lead to starvation or to excessive delays for some tasks. For example, long tasks can suffer with STF if short ones keep arriving. Similarly, tasks of *large* jobs can suffer with LRTF and SRJF. To circumvent this problem, during reordering we check whether a task has waited too long in the queue. If so, we give it higher priority. In particular, we specify both a hard (ST) and a relative (ST_r) threshold. A task is marked as starved if it has waited longer than ST_r times its duration or if it has waited longer than ST secs. ST_r allows short tasks to starve faster than long ones (e.g., a 2-sec task should be marked as starved sooner than a 500-sec task, but not more than ST secs).

4. Global Policies

Our queue management techniques presented so far focused on task execution at specific nodes. We now discuss how Yaq can be coupled with cluster-wide policies. In particular, we first focus on techniques for global job reordering in the case of a centralized design (Section 4.1), and then on imposing sharing policies, such as fairness and capacity (Section 4.2).

4.1 Global Job Reordering

As discussed in Section 2.4, Yaq-c maintains a queue of jobs at the RM.[6] Along with task reordering at each node, we can also devise *job* reordering strategies to be used at the RM. Similar to the task reordering strategies presented in Section 3.3, we can apply SRJF, LRTF and EJF at the job level. More specifically, SRJF will give higher priority to jobs with the smallest remaining work, whereas LRTF will prioritize jobs with the least remaining number of tasks. EJF uses a FIFO queue. The analogous to STF would be SJF (Shortest Job First), assuming we have information about the job durations. Moreover, our starvation-aware techniques can be applied here as well, to avoid jobs from waiting too long in the RM queue. More advanced multi-resource packing techniques (such as Tetris [15]) can also be employed.

Although prioritizing jobs at the RM can be beneficial, what is more interesting in Yaq is how this global job reordering interplays with the local task reordering strategies, as they might have conflicting goals. For instance, when SRJF/LRTF are used both globally and locally, they are expected to further improve job completion times. However, this is probably not the case when SRJF is enabled globally and EJF locally: the former will favor jobs that are close to completion, whereas the latter will locally favor tasks with

[6] Notice that there can be no global job reordering in our distributed Yaq-d design, as there is no global queue in the system.

smaller duration. Our initial results show that there are indeed combinations that can further improve job completion times. We are currently working on formalizing such beneficial combinations, also taking into account workload characteristics.

4.2 Sharing Policies

Scheduling frameworks facilitate sharing of cluster resources among different users by imposing sharing policies. For instance, YARN [27] can impose both fairness (each user gets a fair share of the cluster) [5] and capacity (each user gets a percentage of the cluster) [4] constraints. Sparrow also shows how to impose weighted fair sharing in a distributed setting [22].

All these existing techniques can be applied in Yaq-c and Yaq-d in order to impose sharing constraints *over both running and queued tasks*. However, the scheduling framework has to impose constraints over the actual cluster resources (this is what the user actually observes). When task prioritization is disabled, the sharing constraints over the actual resources will be met, as each task will be executed in the order it was submitted by the scheduler. The problem arises in case of queue reordering: the scheduler has imposed constraints assuming a specific execution order, but this order might change, giving resources to the wrong users, thus exceeding their cluster share against others.

To circumvent this problem, we exploit the starvation threshold ST of our prioritization algorithm (see Section 3.3). In particular, given that each task is marked as starved after ST seconds, actual resources will be given to it and sharing constraints will be met after that period of time.[7] As we experimentally show in Section 6.4, Yaq-c is indeed able to successfully meet strict capacity constraints with only slight momentary violations.

Going further, we make the observation that the above technique is pessimistic in that it does not take advantage of user information about the queued tasks. If two tasks belong to the same user, they are not actually causing violation of sharing constraints between them. This can be solved by pushing auxiliary information about the users to worker nodes. Moreover, it is interesting to investigate whether task prioritization strategies can momentarily allow violations of sharing constraints in order to achieve better job completion times (using some form of deficit counters [24]).

5. Implementation

Yaq-c We implemented Yaq-c by extending Apache Hadoop's YARN [6] as follows. First, we extended YARN's NM to allow local queuing of tasks, and implemented our queue management techniques for bounding queue lengths (Section 3.1) and prioritizing task execution (Section 3.3). Second, we extended YARN's scheduler to enable placement of tasks to queues (Section 3.2), support job prioritization (Section 4.1), and respect cluster sharing constraints in the presence of task queuing (Section 4.2). Finally, in the current implementation, we modified Hadoop's capacity scheduler [4], but our changes can be applied to any Hadoop-compatible scheduler (e.g., DRF [13], fair scheduler [5]).

Yaq-d We implemented Yaq-d by extending the distributed part of Mercury [20, 29] that already supports queuing at worker nodes. In particular, we implemented our techniques for task placement to queues and task prioritization on top of Mercury. In our current implementation, we do not bound the queue lengths, although that could be possible by allowing tasks to be queued at the JMs, in case no queue slots are available in a node. However, as our experimental results show, we already get significant gains over Mercury even without bounding queue lengths.

We have already made available the addition of task queues in YARN's NMs at Apache JIRA YARN-2883 [30]. We also plan to open-source our queue management techniques both on YARN and Mercury.

6. Experimental Evaluation

The main results of our evaluation are the following:

- Yaq-c improves median job completion time (JCT) by 1.7x when compared to YARN over a production workload.
- Yaq-d, when evaluated over the same workload, achieves 9.3x better median JCT when compared to a scheduler that mimics Sparrow's batch sampling, and 3.9x better median JCT when compared to the distributed version of Mercury [20].
- Although task prioritization appears to provide the most pronounced benefits, the combination of all our techniques is the configuration that gives the best results.

Note that our purpose in this work is not to compare Yaq-c with Yaq-d. Instead, we want to study the performance improvement that Yaq-c and Yaq-d bring when compared to existing designs of the same type (centralized and distributed, respectively). Since they follow different architectures, each of them targets different scenarios: high level placement decisions and strict cluster sharing policies for Yaq-c; fast allocation latency and scalability for Yaq-d. Applying our techniques to hybrid schedulers [10, 20] would be an interesting direction for future work.

We now present results from our experimental evaluation. We first assess the performance of both Yaq-c (Section 6.2) and Yaq-d (Section 6.3) over a Hive production workload used at Microsoft, comparing our systems against existing centralized and distributed scheduling schemes. Then we show that Yaq-c can successfully impose sharing invariants (Section 6.4). Lastly, we show a set of micro-experiments that highlight specific components of our de-

[7] As long as task preemption is enabled, otherwise a starved task has to wait for one of the running tasks to finish its execution.

sign, such as queue-bounding, task placement, and task prioritization (Section 6.5).

6.1 Experimental Setup

Cluster setup We deployed Yaq-c and Yaq-d on a cluster of 80 machines and used it for our evaluation. Each machine has a dual quad-core Intel Xeon E5-2660 processor with hyper-threading enabled (i.e., 32 virtual cores), 128 GB of RAM, 10 x 3 TB data drives configured as a JBOD. Inter-machine connectivity is 10 Gbps.

Our Yaq-c implementation is based on YARN 2.7.1. We use the same YARN version to compare against "stock" YARN. The Mercury implementation that we used was based on YARN 2.4.2, and the same holds for Yaq-d, since it was built by extending Mercury, as we explain in Section 5. We also use Tez 0.4.1 [7] to execute all workloads, and Hive 0.13 for the Hive workload that is described below. In all our experiments, we use a heartbeat interval of 3 sec, which is also the typical value used in the YARN clusters at Yahoo! [27].

Workloads To evaluate Yaq-c and Yaq-d against other approaches, we use the Hive-MS workload, which is a Hive [25] workload used by an internal customer at Microsoft to perform data analysis. This is Workload 2 depicted in Figure 1. It consists of 185 queries, each having one map and one reduce phase. The underlying data consists of five relations with a total size of 2.49 PB. Each job has an average of 57.9 mappers and 1.5 reducers. Tasks among all jobs have an average duration of 22.9 sec with a standard deviation of 27.8 sec, when run on stock YARN.

We also use synthetic GridMix [17] workloads, each consisting of 100 tasks/job executed for 30 min, where: (1) X sec is a homogeneous workload where all tasks in a job have the same task duration (e.g., 5 sec), (2) Mixed-5-50 is a heterogeneous workload comprising of 80% jobs with 5-second tasks and 20% jobs with 50-second tasks, and (3) GridMix-MS is another heterogeneous workload, in which task sizes follow an exponential distribution with a mean of 49 sec. GridMix-MS is based on Microsoft's production Workload 1, depicted in Figure 1, after scaling down the longer task durations to adapt them to the duration of our runs and the size of our cluster.

Moreover, in our experiments, the scheduler gets as input the estimated average task duration of the stage (map or reduce) each task belongs to, as observed by previous executions of the same job. Note that we deliberately provide such simple estimates, in order to assess Yaq under imprecise task durations. These estimates are then used during placement of tasks to nodes and for some of our task prioritization algorithms (see also discussion in the beginning Section 3).

Metrics We base our analysis on the following metrics: *job completion time*, which is the time from the moment a job started its execution until the moment all tasks of the job finished execution; *slot utilization*, which is the number of slots[8] occupied at each moment across all machines, divided by the total number of slots in the cluster; *task queuing delay*, which is the time from the moment a task gets placed in a node's queue until its execution starts; *average job throughput*, which is the number of jobs in a workload, divided by the total time needed to execute all jobs, and is used to calculate effective cluster throughput.

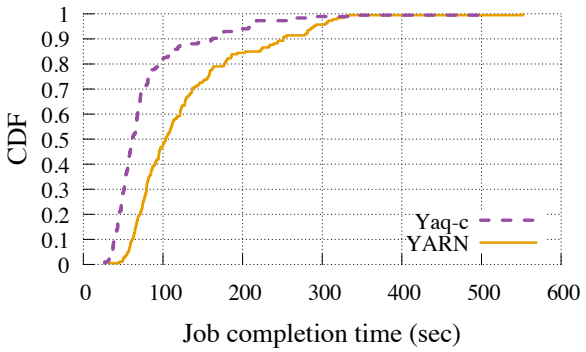

Figure 6. Job completion times for Yaq-c on Hive-MS workload.

	Task queuing delay (sec)			**Job throughput**
	Mean	Stdev	Median	(jobs/min)
Yaq-c	8.5	21.4	1.1	13.9
Yaq-c (unbounded)	65.5	85.1	30.4	5.6
Yaq-c (no reorder)	53.2	78.2	25.4	7.6
YARN	-	-	-	8.8

Table 3. Average task queuing delay and job throughput for Yaq-c on Hive-MS workload.

6.2 Evaluating Yaq-c

To evaluate Yaq-c, we compare it against stock YARN. For Yaq-c we use a queue size of four slots (Section 3.1), the queue wait time-based placement policy (Section 3.2) and the SRJF prioritization policy (Section 3.3), as those gave us the best results. The queue size we used coincides with the one suggested by our analysis using Equation 1. Figure 6 shows that Yaq-c achieves better job completion times across all percentiles with a 1.7x improvement for median job completion time. As shown in Table 3, Yaq-c also improves job throughput by 1.6x over YARN. These gains are due to the higher cluster utilization Yaq-c achieves by having worker-side queues (more details on utilization are given in Section 6.5.1). Moreover, to show the benefit of our queue management techniques, in Table 3 we provide performance numbers for Yaq-c if we disable queue length bounding or task prioritization. In the absence of our techniques, we observe excessive task queuing delays that negatively impacts job throughput, also resulting in worse performance than

[8] We use 4 GB and 1 CPU per slot.

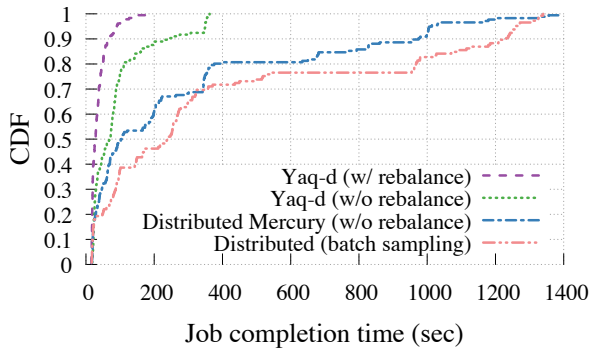

Figure 7. Job completion times for Yaq-d on Hive-MS workload.

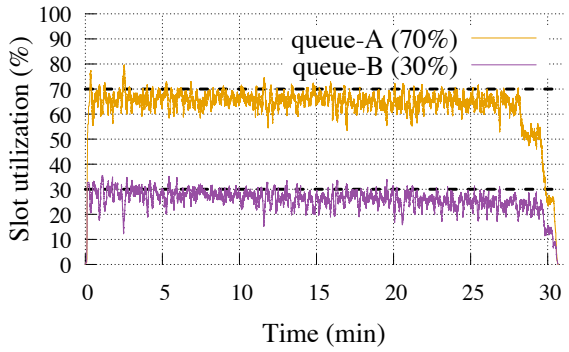

Figure 8. Imposing capacity constraints in Yaq-c.

	Task queuing delay (sec)			**Job throughput**
	Mean	Stdev	Median	(jobs/min)
Yaq-d (w/ rebalance)	17.9	54.2	0.35	16.6
Yaq-d (w/o rebalance)	34.2	67.0	5.6	10.1
Distributed Mercury	49.7	73.7	12.9	5.8
Distributed (batch sampl.)	81.4	101.4	26.2	5.3

Table 4. Average task queuing delay and job throughput for Yaq-d on Hive-MS workload.

stock YARN. On the contrary, Yaq-c achieves a median task queuing delay of only 1.1 sec.

6.3 Evaluating Yaq-d

We evaluate Yaq-d against two other distributed scheduler variants: *distributed Mercury*, which uses the distributed part of Mercury [20], and *distributed batch sampling*, for which we modified Mercury to perform task placement using batch sampling, as a way to simulate the task placement done by Sparrow [22]. We use two different Yaq-d configurations with and without dynamic queue rebalancing (see end of Section 3.2). Moreover, we use the queue wait time-based placement policy (Section 3.2) and the SRJF prioritization policy with a 10-sec hard starvation threshold (Section 3.3), which performed best in practice.

Our results for the Hive-MS workload are depicted in Figure 7 and Table 4. Yaq-d (with rebalance) improves job completion time (JCT) across all percentiles when compared to both Mercury and batch sampling. In particular, it achieves better median JCT by 3.9x over Mercury and by 9.3x over batch sampling. These improvements are due to the efficient management of the local queues, as we significantly reduce the task queuing delays and thus the head-of-line blocking.

Observe that in our Yaq-d implementation we do not use late binding of tasks to nodes, as it conflicts with some of YARN's design choices. As shown in Figure 8 of the Sparrow paper [22], late binding on top of batch sampling further improves average job completion time by 14% and the 95th percentile by ∼30%. Therefore, even if we implemented late binding, most probably Yaq-d would still significantly outperform Sparrow.

As can be seen from Table 4, Yaq-d also achieves a higher job throughput by 2.9x over Mercury and by 3.1x over batch sampling. When configuring Mercury and batch sampling, we had to tune the number of jobs that are allowed to be executed concurrently: allowing too many concurrent jobs improves job throughput but hurts JCT (due to having tasks belonging to many different jobs being queued at the nodes without properly sizing or reordering the queues); allowing fewer jobs improves JCT but leads to lower utilization and hurts job throughput. We could improve job throughput for Mercury and batch sampling in our runs by allowing more concurrent jobs, but that would lead to even worse JCT. On the contrary, Yaq-d improves both JCT and job throughput at the same time.

6.4 Imposing Sharing Constraints

As discussed in Section 4.2, task prioritization can potentially lead to violation of cluster-wide sharing policies. To this end, we use Yaq-c, whose implementation relies on Hadoop's capacity scheduler [4] (as explained in Section 5) that is capable of imposing capacity quotas on each user of the cluster. To investigate whether Yaq-c continues to respect such cluster-wide sharing policies despite task prioritization, we configure the capacity scheduler with two queues, A and B, where the cluster capacity is split 70% and 30% respectively. We run a GridMix workload that submits jobs to both queues with equal probability. Figure 8 shows cluster-wide slot utilization for each of these two queues measured from the perspective of all worker nodes. As the figure shows, Yaq-c respects each queue's capacity with only some momentary slight violations.

6.5 Micro-experiments

We evaluate specific aspects of our queue management techniques using a set of micro-experiments. In these runs we use our synthetic GridMix workloads, which make it easier to experiment with different task duration distributions, whenever needed. We study the effects of bounding queuing lengths (Section 6.5.1), task placement choices (Section 6.5.2), and task prioritization strategies (Section 6.5.3).

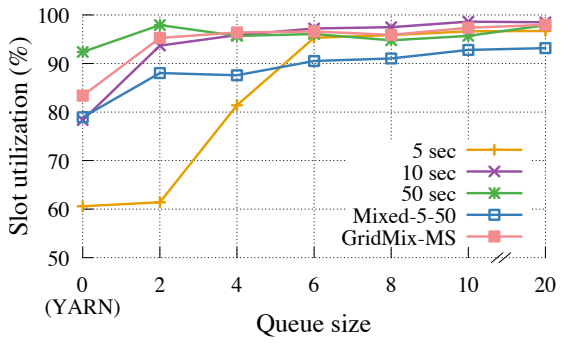

Figure 9. Average cluster slot utilization with different workloads and queue lengths.

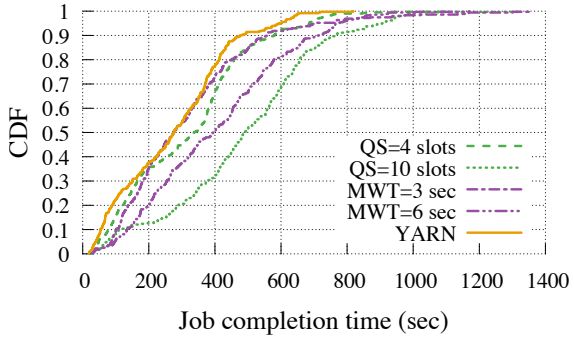

Figure 10. Job completion time for GridMix-MS workload with different queue bounding techniques and no task prioritization.

We also evaluate our techniques over a heavy-tailed distribution (Section 6.5.4). Here we use Yaq-c, but we also observed similar trends with Yaq-d for task placement and prioritization.

6.5.1 Bounding Queuing Delays

We first study the impact of queue length in cluster utilization and job completion times (JCT). To this end, we purposely disable task prioritization in these experiments.

Figure 9 shows how slot utilization for Yaq-c varies for different workloads when introducing queuing at worker nodes. By masking feedback delays between the RM and NM, Yaq-c is able to prevent slots from becoming idle. The gains are particularly pronounced when task durations are short: for 5-sec tasks, average utilization is 60% with YARN but goes up to 96% with Yaq-c. The graphs also show that utilization improves with longer queue sizes, as expected. Furthermore, once the nodes are saturated increasing the queue sizes even further does not improve utilization. For instance, the 5 sec workload needs a queue size of six slots to achieve full utilization, while for the 50 sec workload a queue size of two slots is sufficient.

Figure 10 compares job completion time (JCT) of the GridMix-MS workload with YARN and both length-based

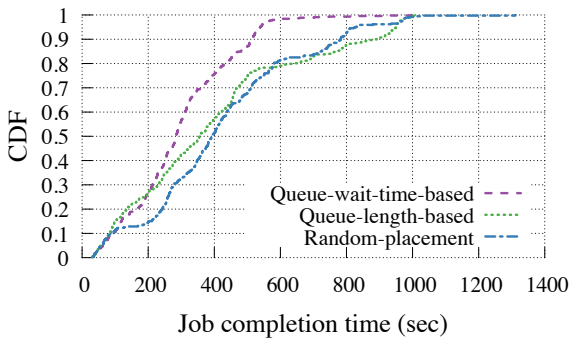

Figure 11. Job completion time for the GridMix-MS workload with different RM task placement policies.

(QS=x denotes that x tasks are allowed to be queued) and delay-based bounding (MWT=x denotes that queuing delay should not exceed x sec). For fixed queue lengths, we see that JCT increases with queue length. This is to be expected since increased queue lengths introduce higher queuing delays, without further improving utilization (as shown in Figure 9). Furthermore, the tail of the distribution also increases substantially when queue lengths increase, by upwards of 1.7x for MWT=3 compared to YARN.

Figure 9 and Table 3 reveal that *simple queues at worker nodes, even if bounded, negatively impact job completion times most of the time.* Only in a small number of cases, for some homogeneous workloads, we saw improvement in JCT just by bounding queue lengths. However, as we show in Table 3 and later in Section 6.5.3, queue bounding coupled with task prioritization brings significant JCT gains.

6.5.2 Task Placement at Queues

We now compare different task placement strategies. We use our two strategies, namely queue length-based and queue wait time-based placement (see Section 3.2), as well as a random placement strategy that randomly assigns tasks to nodes. We use a fixed queue size of six slots with task prioritization disabled. Job completion times for these runs are shown in Figure 11. As expected, the placement that is based on queue wait time outperforms the rest of the strategies, since it uses richer information about the status of the queues. In particular, it improves median job completion time by 1.2x when compared to the queue length-based and by 1.4x to the random strategy. Also note that the random placement has a significantly longer tail than our two strategies. Therefore, in all our experiments we use the queue wait time-based placement.

6.5.3 Task Prioritization at Worker Nodes

Figure 12 shows the job completion times (JCT) for our three task reordering algorithms (LRTF, SRJF, STF). We use a queue length of ten slots (unless otherwise stated) and no hard or relative starvation thresholds.

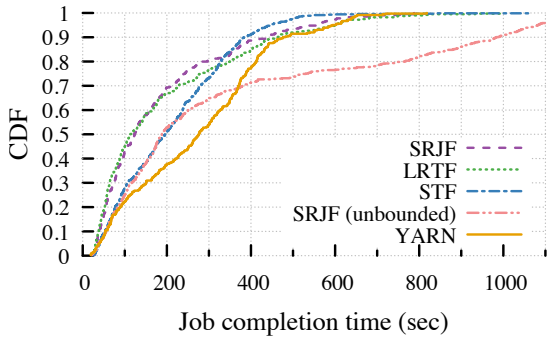

Figure 12. Job completion time for GridMix-MS workload with different task prioritization algorithms.

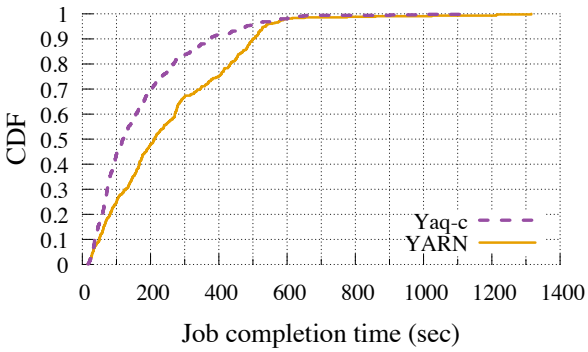

Figure 14. Job completion time for the heavy-tailed workload that is based on GridMix-MS.

Figure 13. Job completion time for the GridMix-MS workload with different hard starvation thresholds (ST) and STF prioritization.

The job-aware policies SRJF and LRTF perform the best when compared to YARN: 2.2x better median JCT for SRJF and 2.4x for LRTF. The non job-aware STF reordering policy performs 1.4x better than YARN. The difference in performance between STF and the other methods is that STF is more aggressive than others in attempting to fix head-of-line blocking issues, but can quickly lead to starvation issues (which are addressed later in this section). Thus, job progress is a much more reliable metric to use when reordering than the local metrics STF uses. Interestingly, for the GridMix workload LRTF performed better than SRJF (most probably due to the predictability of the synthetic workload). However, in the real Hive-MS workload, SRJF worked best.

In the same figure, we have included a run with SRJF prioritization and no queue bounding (marked "unbounded"). This run shows that with queue bounding disabled, task prioritization improves the lower percentiles of JCT, but negatively impacts the higher ones. Based also on the results of Table 3, it becomes clear that *combining task prioritization with queue bounding is required to get the best results in terms of JCT.*

Starvation Threshold We perform various runs to study the impact our starvation thresholds (see Section 3.3) have on the performance of Yaq-c. The hard starvation threshold (ST) and relative starvation threshold (ST_r) both provide the ability to limit how long a task is starved. We empirically found ST_r to provide less benefit in decreasing overall job completion time (JCT) when compared to the effects of ST. The results we present here showcase the effect of various hard starvation limits for the STF reordering policy, which benefits the most from the starvation parameter (given it is not job-aware as we discussed above). Figure 13 shows JCT with the GridMix-MS workload using STF reordering, a fixed queue size of ten slots, and various ST values. First, we observe that STF is sensitive to the ST value that is used. A value of 0.5 sec, which marks tasks as starved early, essentially falling back to the EJF strategy, works best for this synthetic workload with tasks of each job being relatively homogeneous. High values (ST=10 sec) are detrimental, whereas a value of 1.5 sec improves JCT for some of the jobs. Our experiments also revealed that SRJF and LRTF reordering are less sensitive to different ST values and that relatively higher values can give better results. Being job-aware, these strategies already prioritize the execution of starved straggler tasks. For instance, an ST value of 10 sec worked best on the more realistic Hive-MS production workload with SRJF. This also suggests that the ST value should be calibrated based on the characteristics of the workloads and the used strategy.

6.5.4 Heavy-tailed Workload

The task durations of the heterogeneous workloads we have used so far (GridMix-MS and Hive-MS) follow an exponential distribution. In order to assess the impact of our techniques on workloads with different characteristics, we modified GridMix-MS so that its task durations follow a heavy-tailed distribution. Specifically, we increased the duration of the longest 20% tasks by 500 secs. We use Yaq-c with a queue length of ten slots and the SRJF reordering strategy. Figure 14 show the gains in JCT that Yaq-c yields for this heavy-tailed workload. In particular, it improves median job completion time by 1.8x when compared with YARN.

7. Related Work

Our focus in Yaq is on the effective management of local task queues at worker nodes, and, as such, is complementary to extensive prior work in centralized [14, 18, 26–28], distributed [8, 11, 22, 23], and hybrid [10, 20] cluster schedulers. We covered many aspects of these systems in previous sections (particularly, in Section 1 and Section 2), and complement our discussion here.

Local queues exist by necessity in distributed schedulers, such as Sparrow [22], Tarcil [11], Hawk [10], Apollo [8], and Mercury [20]. Sparrow and Hawk rely on the power of two choices balancing technique when placing tasks. Tarcil extends Sparrow's placement by adopting a dynamically adjusted sample size. On the other hand, in Yaq-d, similar to Apollo and Mercury, each scheduler uses information about the nodes' status to perform more educated placement decisions, which is crucial for heterogeneous workloads.

To the best of our knowledge, in all existing schedulers, whenever a running task finishes the next task selected for execution is mostly based on FIFO. Apollo acknowledges that queues can go beyond FIFO and be reordered, but does not explore this in depth. In contrast, we present the first extensive study of the impact of different queue management strategies in the cluster's performance.

While simple to implement, FIFO ordering can cause head-of-line blocking whenever task execution times differ significantly. This in turn impacts predictability of job execution times. To mitigate this issue, existing systems take extensive corrective mechanisms, such as duplicate scheduling [8], dynamically rebalancing queues across nodes [20], work stealing [10], and straggler mitigation [2, 12, 31]. Since head-of-line blocking issues are inherent to queuing systems, similar to these systems, Yaq also incorporates corrective mechanisms. Yaq goes beyond these systems by avoiding these problems in the first place via careful placement of tasks to nodes, bounding of queues and prioritization of task execution, thus improving job completion times.

Our task prioritization strategies (see Section 3.3) have commonalities with multiprocessor scheduling [3]. For instance, SRJF is similar to the Shortest Remaining Time First (SRTF) scheduling algorithm. However, unlike OS scheduling, SRJF relies on job progress information arriving from the RM/UM periodically, which can be stale. Moreover, in Yaq we can only perform local reordering of tasks once they have already been dispatched to a worker node.

Finally, our queue management techniques are related to the scheduling of packet flows in networks. The goal in network scheduling is to find a sweet spot between bandwidth utilization and flow completion time, which can be seen as related to cluster utilization and job completion time in cluster scheduling, respectively. Recent work like, PDQ [19] schedules flows based on earliest deadline first, pFabric [1] relies on remaining flow size, and DeTail [33] on application priorities. QJump [16] prioritizes packets based on flow classes, set by a network administrator.

8. Conclusion

Our work is motivated by the observation that choosing between existing cluster scheduling frameworks imposes an unnecessary trade-off. On one hand centralized schedulers favor predictable execution at the expense of utilization; on the other hand, distributed schedulers improve cluster utilization but suffer from high job completion time when workloads are heterogeneous. To address this trade-off, we built around the idea of introducing queues at worker nodes. In particular, a novel contribution of our work is that by employing queues for centralized frameworks, we achieve utilization comparable to distributed schemes. We then develop policies for active queue management, carefully choosing which task to execute next whenever a running task exits, with the goal of fast job completion times. The policies we develop are equally applicable to both centralized and distributed scheduling frameworks. We built Yaq as an extension to YARN, deployed it on a large cluster and experimentally demonstrated the gains using production as well as synthetic workloads. Yaq improves job completion time across all percentiles and, in particular, median job completion time by up to 9.3x, when compared to existing scheduling schemes, such as YARN, Mercury and an implementation of Sparrows batch sampling on Mercury.

Acknowledgments

We would like to thank our shepherd, Malte Schwarzkopf, for his detailed feedback that helped improve the paper, as well as the anonymous reviewers for their valuable comments. We also thank Carlo Curino, Chris Douglas, Subru Krishnan, Ishai Menache, Arun Suresh, Chuck Thacker and George Varghese for the fruitful discussions. Jeff Rasley is supported by an NSF Graduate Research Fellowship (DGE-1058262).

References

[1] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *SIGCOMM*, 2013.

[2] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, 2010.

[3] T. Anderson and M. Dahlin. *Operating Systems: Principles and Practice*. Recursive Books, second edition, 2014.

[4] Apache Hadoop Capacity Scheduler. `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`.

[5] Apache Hadoop Fair Scheduler. `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html`.

[6] Apache Hadoop Project. http://hadoop.apache.org/.

[7] Apache Tez. https://tez.apache.org.

[8] 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.

[9] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013. URL http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext.

[10] P. Delgado, F. Dinu, A. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *USENIX ATC*, 2015.

[11] C. Delimitrou, D. Sanchez, and C. Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *SoCC*, 2015.

[12] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *EuroSys*, 2012.

[13] 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.

[14] A. Goder, A. Spiridonov, and Y. Wang. Bistro: Scheduling data-parallel jobs against live production systems. In *USENIX ATC*, 2015.

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

[16] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues Don't Matter When You Can JUMP Them! In *NSDI*, 2015.

[17] Hadoop GridMix. http://hadoop.apache.org/docs/r1.2.1/gridmix.html.

[18] 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.

[19] C. Hong, M. Caesar, and B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM*, 2012.

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

[21] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters. In *HotOS*, 2013.

[22] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, 2013.

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

[24] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round-robin. *IEEE/ACM Trans. Netw.*, 1996.

[25] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2), 2009.

[26] Under the Hood: Scheduling MapReduce jobs more efficiently with Corona. http://tinyurl.com/fbcorona.

[27] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*, 2013.

[28] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[29] YARN-2877. Extend YARN to support distributed scheduling. https://issues.apache.org/jira/browse/YARN-2877.

[30] YARN-2883. Queuing of container requests in the NM. https://issues.apache.org/jira/browse/YARN-2883.

[31] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.

[32] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[33] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. H. Katz. DeTail: Reducing The Flow Completion Time Tail In Datacenter Networks. In *SIGCOMM*, 2012.

A. Determining the Queue Length

We consider an arbitrary node that has r run slots and a queue of length b slots. We want to determine the value of the parameter b such that the utilization of the node is at least $1-\delta$ for given parameter $\delta \in (0,1]$.

We admit the following assumptions. Let τ be the length of a heart-beat interval. The node is fed with new tasks at the beginning of each heart-beat interval such that there are at most r tasks being processed by the node and at most b tasks being queued for processing at the node. Whenever the node completed processing a task it starts processing one of the tasks from the queue taken uniformly at random, if there is any in the queue. We assume that task processing times are independent and identically distributed according to exponential distribution with mean $1/\mu$. This assumption enables us to characterize the node utilization by leveraging the memory-less property of the exponential distribution.

The node *utilization* is denoted with u and is defined as the average fraction of time the run slots of the node are busy processing tasks over an asymptotically large time interval. More formally, let $Q_i(t) = 1$ if at time t run slot i is busy, and $Q_i(t) = 0$, otherwise. Then, the node utilization is defined by

$$u = \lim_{T\to\infty} \frac{1}{T} \int_0^T \frac{\sum_{i=1}^{r} \mathbf{1}(Q_i(t)=1)}{r} dt.$$

where $\mathbf{1}(A) = 1$ if condition A is true, and $\mathbf{1}(A) = 0$, otherwise.

Let $X_{n,\lambda}$ be a random variable with distribution that corresponds to the sum of n independent random variables with exponential distribution of mean $1/\lambda$. Note that the distribution of $X_{n,\lambda}$ is Erlang distribution with parameters n and λ, wich has the density function

$$f_{n,\lambda}(x) = \frac{\lambda^n x^{n-1} e^{-\lambda x}}{(n-1)!}, \text{ for } x \geq 0.$$

Proposition 2. *Under the given assumptions, the node utilization is given by*

$$\begin{aligned} u &= 1 - \left(\left(1 - \frac{1}{\mu\tau}\right) \Pr[X_{b,r\mu} \leq \tau] \right. \\ &\quad \left. + \frac{1}{\mu\tau} \frac{e^{-\mu\tau}}{\left(1 - \frac{1}{r}\right)^b} \Pr[X_{b,(r-1)\mu} \leq \tau] \right) \end{aligned}$$

Proof. We are interested in the node utilization with respect to the stationary distribution. Suppose that time 0 is the beginning of a heart-beat interval. We can use the Palm inversion formula (or "cycle formula") to note that the node utilization is equal to

$$u = \frac{\sum_{i=1}^{r} \mathbb{E}\left[\int_0^\tau \mathbf{1}(Q_i(t) = 1) dt\right]}{r\tau}.$$

It suffices to consider an arbitrary run slot i of the node and characterize the expected value of $\int_0^\tau \mathbf{1}(Q_i(t) = 1)dt$. By the memory-less property of the exponential distribution, there are $r + b$ tasks at time 0 whose (residual) processing times are independent and have exponential distribution with mean $1/\mu$. Whenever there are r tasks being processed by the node, the earliest time until completion of a task is equal in distribution to a minimum of r independent exponentially distributed random variables each with mean $1/\mu$; hence, it has exponential distribution with mean $1/(r\mu)$. It follows that the earliest time at which the queue is empty is equal in distribution to $X_{b,r\mu}$. From this time instance, each run slot completes processing the task assigned to it after an independent random duration that has exponential distribution with mean $1/\mu$. From this discussion, we conclude that

$$\begin{aligned} &\mathbb{E}\left[\int_0^\tau \mathbf{1}(Q_i(t) = 1)dt\right] = \tau \Pr[X_{b,r\mu} > \tau] \\ &+ \int_0^\tau \mathbb{E}[\min\{\sigma, \tau - x\}] d \Pr[X_{b,r\mu} \leq x] \end{aligned}$$

where σ is a random variable with exponential distribution with mean $1/\mu$.

By a simple calculus, we have

$$\begin{aligned} \mathbb{E}[\min\{\sigma, t\}] &= \int_0^\infty \Pr[\min\{\sigma, t\} > x] dx \\ &= \int_0^t \Pr[\sigma > x] dx \\ &= \int_0^t e^{-\mu x} dx \\ &= \frac{1}{\mu}\left(1 - e^{-\mu t}\right). \end{aligned}$$

Hence, it follows that the utilization is given by

$$\begin{aligned} u &= \Pr[X_{b,r\mu} > \tau] \\ &+ \frac{1}{\mu\tau} \int_0^\tau \left(1 - e^{-\mu(\tau - x)}\right) d \Pr[X_{b,r\mu} \leq x] \end{aligned}$$

which by some elementary calculus can be written as asserted in the proposition. □

Notice that, in particular, for a node with zero queue slots,

$$u = \frac{1 - e^{-\mu\tau}}{\mu\tau}.$$

A simple lower bound on the node utilization can be derived as follows. Let A_k denote the event that in the k-th heart-beat interval none of the run slots is every idle. Notice that

$$u \geq \Pr[A_k].$$

The event A_k is equivalent to the event that the time elapsed from the k-th heart beat until the completion of the $(b+1)$-st task, among the tasks present just after the k-th heart beat, is larger than the length of the heart-beat interval τ. Notice that the distribution of this time duration is equal Erlang distribution with parameters $b+1$ and $r\mu$. Hence, we have

$$\Pr[A_k] = \Pr[X_{b+1,r\mu} > \tau].$$

It follows that a sufficient condition for the node utilization to be at least $1 - \delta$ is the following condition

$$\Pr[X_{b+1,r\mu} \leq \tau] \leq \delta. \tag{2}$$

Proposition 3. *A sufficient condition for the probability that in a heart-beat interval none of the run slots is ever idle is at least $1 - \delta$ is that the queue length b is the smallest integer such that it holds*

$$r\mu\tau \left(1 + \frac{b+1}{r\mu\tau}\left(\log\left(\frac{b+1}{r\mu\tau}\right) - 1\right)\right) \geq \log\left(\frac{1}{\delta}\right). \tag{3}$$

Before giving a proof of the proposition, we discuss the asserted sufficient condition. If the task processing times were deterministic assuming a common value $1/\mu$ and the length of the heart-beat interval is a multiple of $1/\mu$, then for 100% utilization it is necessary and sufficient to set the queue length such that $b + r = r\mu\tau$. This yields the queue length that is linear in $r\mu\tau$, for any fixed value of the run slots r. The sufficient condition in (3) requires a larger queue length than $r\mu\tau$ for small values of $r\mu\tau$. It can be shown that

the sufficient condition (3) requires the queue length that is at least $r\mu\tau + \sqrt{\log(1/\delta)}\sqrt{r\mu\tau}$, for large $r\mu\tau$.

For numerical examples, see Figure 5. Specifically, given a heartbeat interval $\tau = 3$ sec, an average task duration $1/\mu$ of 10 sec, $r = 10$ tasks allowed to be executed at a node at the same time, and a target utilization of 95%, a queue of $b = 6$ slots is required. Likewise, for an average task duration of 30 sec, the queue size should be ≥ 3 slots. These values for b are also validated by our experiments (Section 6) on the production Workload 2 of Figure 1.

Proof. The proof follows by (2) and the Chernoff's inequality, which we describe as follows.

We first establish the following claim:

$$\Pr[X_{n,\lambda} \leq x] \leq e^{-\lambda x\left(1+\frac{n}{\lambda x}\left(\log\left(\frac{n}{\lambda x}\right)-1\right)\right)}, \text{ for } x \geq 0. \quad (4)$$

Let $\sigma_1, \sigma_1, \ldots, \sigma_n$ be a sequence of independent exponentially distributed random variables each of mean $1/\lambda$. Using Chernoff's inequality, for every $\theta > 0$, we have

$$\begin{aligned} \Pr[X_{n,\lambda} \leq x] &\leq e^{\theta x}\mathbb{E}[e^{-\theta\sum_{i=1}^{n}\sigma_i}] \\ &= e^{\theta x}\prod_{i=1}^{n}\mathbb{E}[e^{-\theta\sigma_i}] \\ &= e^{\theta x}\mathbb{E}[e^{-\theta\sigma_1}]^n \\ &= e^{\theta x}\left(\frac{\lambda}{\lambda+\theta}\right)^n. \end{aligned}$$

The minimizer of the last expression is for the value of parameter θ such that

$$\lambda x + \theta x = n.$$

Hence, we obtain the inequality asserted in (4).

Using (4), have

$$\Pr[X_{b+1,r\mu} \leq \tau] \leq e^{-r\mu\tau\left(1+\frac{b+1}{r\mu\tau}\left(\log\left(\frac{b+1}{r\mu\tau}\right)-1\right)\right)}.$$

By requiring that the right-hand side in the last inequality is smaller than or equal to δ, we obtain the inequality asserted in the proposition.

For every integer value b such that condition (3) holds, we have that $\Pr[X_{b+1,r\mu} \leq \tau] \leq \delta$, which implies the node utilization of at least $1 - \delta$. Since the left-hand side of the inequality in (3) is increasing in b, it suffices to chose the queue length that is the smallest integer b such that condition (3) holds. □