Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters

Konstantinos Karanasos  Sriram Rao  Carlo Curino  Chris Douglas  Kishore Chaliparambil  Giovanni Matteo Fumarola  Solom Heddaya  Raghu Ramakrishnan  Sarvesh Sakalanaga

Microsoft Corporation

Abstract

Datacenter-scale computing for analytics workloads is increasingly common. High operational costs force heterogeneous applications to share cluster resources for achieving economy of scale. Scheduling such large and diverse workloads is inherently hard, and existing approaches tackle this in two alternative ways: 1) centralised solutions offer strict, secure enforcement of scheduling invariants (e.g., fairness, capacity) for heterogeneous applications, 2) distributed solutions offer scalable, efficient scheduling for homogeneous applications.

We argue that these solutions are complementary, and advocate a blended approach. Concretely, we propose Mercury, a hybrid resource management framework that supports the full spectrum of scheduling, from centralised to distributed. Mercury exposes a programmatic interface that allows applications to trade-off between scheduling overhead and execution guarantees. Our framework harnesses this flexibility by opportunistically utilizing resources to improve task throughput. Experimental results on production-derived workloads show gains of over 35% in task throughput. These benefits can be translated by appropriate application and framework policies into job throughput or job latency improvements. We have implemented and contributed Mercury as an extension of Apache Hadoop / YARN.1

1 Introduction

Over the past decade, applications such as web search led to the development of datacenter-scale computing, on clusters with thousands of machines. A broad class of data analytics is now routinely carried out on such large clusters over large heterogeneous datasets. This is often referred to as “Big Data” computing, and the diversity of applications sharing a single cluster is growing dramatically for various reasons: the consolidation of clusters to increase efficiency, the diversity of data (ranging from relations to documents, graphs and logs) and the corresponding diversity of processing required, the range of techniques (from query processing to machine learning) being increasingly used to understand data, the ease of use of cloud-based services, and the growing adoption of Big Data technologies among traditional organizations.

This diversity is addressed by modern frameworks such as YARN [27], Mesos [16], Omega [24] and Borg [28], by exposing cluster resources via a well-defined set of APIs. This facilitates concurrent sharing between applications with vastly differing characteristics, ranging from batch jobs to long running services. These frameworks, while differing on the exact solution (monolithic, two-level or shared-state) are built around the notion of centralized coordination to schedule cluster resources. For ease of exposition, we will loosely refer to all such approaches as centralised scheduler solutions. In this setting, individual per-job (or per-application framework) managers petition the centralized scheduler for resources via the resource management APIs, and then coordinate application execution by launching tasks within such resources.

Ostensibly, these centralized designs simplify cluster management in that there is a single place where scheduling invariants (e.g., fairness, capacity) are specified and enforced. Furthermore, the central scheduler has cluster-wide visibility and can optimize task placement along multiple dimensions (locality [31], packing [15], etc.).

However, the centralized scheduler is, by design, in the critical path of all allocation decisions. This poses scalability and latency concerns. Centralized designs rely on heartbeats which are used for both liveness and for triggering allocation decisions. As the cluster size scales, to minimize heartbeat processing overheads, operators are forced to lower the heartbeat rate (i.e., less frequent heartbeats). In turn, this increases the scheduler’s...
allocation latency. This compromise becomes problematic if typical tasks are short [22]. A workload analysis from one of the production clusters at Microsoft also suggests that shorter tasks are dominant. This is shown as a CDF of task duration in Figure 1. Note that almost 60% of the tasks complete execution under 10 seconds. Therefore, the negative effects of centralized heartbeat-based solutions range from poor latency for interactive workloads to utilization issues (slow allocation decisions means resources are fallow for longer periods of time).

To amortize the high scheduling cost of centralized approaches, the “executor” model has been proposed [29, 19, 21, 22]. This hierarchical approach consists in reusing containers assigned by the central scheduler to an application framework that multiplexes them across tasks/queries.2 Reusing containers assumes that submitted tasks have similar characteristics (to fit in existing containers). Moreover, since the same system-level process is shared across tasks, the executor model has limited applicability to within a single application type. It is, thus, orthogonal to our work.

Fully distributed scheduling is the leading alternative to obtain high scheduling throughput. A practical system leveraging this design is Apollo [9]. Apollo allows each running job to perform independent scheduling choices and to queue its tasks directly at worker nodes. Unfortunately, this approach relies on a uniform workload (in terms of application type), as all job managers need to run the same scheduling algorithm. In this context, allowing arbitrary applications, while preventing abuses and strictly enforcing capacity/fairness guarantees, is non-trivial. Furthermore, due to lack of global view of the cluster, distributed schedulers make local scheduling decisions that are often not globally optimal.

In Figure 2, we pictorially depict the ideal operational point of these three approaches: centralized [16, 27], distributed [9], and executor-model [29, 22], as well as the target operational point for our design. A detailed discussion of related work is deferred to § 8.

2By containers we refer to the allocation units that may comprise multiple resources, such as memory and CPU.

The key technical challenge we explore in this paper is the design of a resource management infrastructure that allows us to simultaneously: (1) support diverse (possibly untrusted) application frameworks, (2) provide high cluster throughput with low-latency allocation decisions, and (3) enforce strict scheduling invariants (§ 2).

Below we present the main contributions of this paper. First: we propose a novel hybrid resource management architecture. Our key insight is to offload work from the centralized scheduler by augmenting the resource management framework to include an auxiliary set of schedulers that make fast/distributed decisions (see Fig. 3). The resource management framework comprising these schedulers is now collectively responsible for all scheduling decisions (§ 3).

Second: we expose this flexibility to the applications by associating semantics with the type of requested containers (§ 3.2). Applications may now choose to accept high scheduling costs to obtain strong execution guarantees from the centralized scheduler, or trade strict guarantees for sub-second distributed allocations. Intuitively, opportunistic jobs or applications with short tasks can benefit from fast allocations the most.

Third: we leverage the newly found scheduling flexibility to explore the associated policy space. Careful policy selection allows us to translate the faster scheduling decisions into job throughput or latency gains (§ 4 and § 5).

Fourth: we implement, validate and open-source this overall design in a YARN-based system called Mercury (§ 6). We compare Mercury with stock YARN by running synthetic and production-derived workloads on a 256-machine cluster. We show 15 to 45% task throughput improvement, while maintaining strong invariants for the applications that need them. We also show that by tuning our policies we can translate these task throughput gains to improvements of either job latency or throughput (§ 7).

The open-source nature [6] and architectural generality of our effort makes Mercury an ideal substrate for other researchers to explore centralized, distributed and hybrid scheduling solutions, along with a rich policy space. We describe ongoing work in § 9.
2 Requirements

Given a careful analysis of production workloads at Microsoft, and conversations with cluster operators and users, we derive the following set of requirements we set out to address with Mercury:

- **R1** Diverse application frameworks: Allow arbitrary user code (as opposed to a homogeneous, single-app workload).
- **R2** Strict enforcement of scheduling invariants: Example invariants include fairness and capacity; this includes policing/security to prevent abuses.
- **R3** Maximize cluster utilization and throughput: Higher cluster utilization and throughput lead to higher return on investment (ROI).
- **R4** Fine-grained resource sharing: Tasks from different jobs can concurrently share a single node.
- **R5** Efficiency and scalability of scheduling: Support high rate of scheduling decisions.

Note that classical centralized approaches target R1-R4, while distributed approaches focus on R3-R5. We acknowledge the tension between conflicting requirements (R2 and R5), each emerging from a subset of the applications we aim to support. In Mercury, we balance this tension by blending centralized and distributed decision-making in a request-specific manner.

**Non-goals** Low latency for sub-second interactive queries is outside the scope of our investigation. This is the target of executor-model approaches [29, 19, 21, 22], which achieve millisecond start times by sharing processes. This is at odds with requirements R1-R2.

3 Mercury Design

We first provide an overview of the Mercury architecture (§ 3.1). Next, we describe the programming interface that Job Managers use for requesting resources (§ 3.2), and how the framework allocates them (§ 3.3). Then we provide details about task execution (§ 3.4).

3.1 Overview

Mercury comprises two subsystems, as shown in Fig. 3:

- **Mercury Runtime**: This is a daemon running on every worker node in the cluster. It is responsible for all interactions with applications, and for the enforcement of execution policies on each node.
- **Mercury Resource Management Framework**: This is a subsystem that includes a central scheduler running on a dedicated node, and a set of distributed schedulers running on (possibly a subset of) the worker nodes, which loosely coordinate through a Mercury

![Mercury Resource Management Framework](image-url)
that priorities are enforced, the runtime can also decide to kill or preempt running tasks (step 7), to allow immediate execution of higher priority tasks.

3.2 Resource Request

When requesting containers, a Job Manager uses Mercury’s programming interface to specify the type of containers it needs. This specification is based on the container’s allocation/execution semantics. Our design defines the following two container classes:

GUARANTEED containers incur no queuing delay, i.e., they are spawned by the Mercury Runtime as soon as they arrive to a worker node. Moreover, these containers run to completion bar failures, i.e., they are never preempted or killed by the infrastructure.

QUEUEABLE containers enable the Job Manager to "queue" a task for execution on a specific node. No guarantees are provided on the queuing delay, or whether the container will run to completion or be preempted.

3.3 Container Allocation

In our design (see Figure 3), GUARANTEED containers are allocated by the central scheduler and QUEUEABLE containers are allocated by one of the distributed schedulers. Requests for either containers are routed appropriately by the Mercury Runtime. Furthermore, both schedulers are free to allocate containers on any node in the cluster. In what follows, we describe the design rationale.

The central scheduler has knowledge about container execution as well as resource availability on individual machines. This information is part of the periodic heartbeat messages that are exchanged between the framework components. Consequently, the central scheduler can perform careful placement of GUARANTEED containers without causing resource contention.

To support fast container allocation, a distributed scheduler restricts itself to allocating QUEUEABLE containers, which can be placed on any machine in the cluster. The distributed scheduler uses lightweight cluster load information, provided by the Mercury Coordinator, for making placement decisions.

The path not taken: We considered and discarded two alternative designs. First the central scheduler could make all scheduling decisions, including QUEUEABLE. Such design would overload the central scheduler. This would be coped with by limiting the rate at which Job Managers can petition the framework for resources (e.g., every few seconds instead of in the millisecond range as we enable with Mercury). This is akin to forfeiting R5. The second alternative sees the framework-level distributed scheduler making all decisions, including GUARANTEED. This would require costly consensus building among schedulers to enforce strict invariants, or relax our guarantees, thus forfeiting R2.

The hybrid approach of Mercury allows us to meet requirements R1- R5 of § 2, as we validate experimentally.

3.4 Task Execution

As described above, Mercury’s centralized and distributed schedulers independently allocate containers on a single shared pool of machines. This in turn means that conflicting allocations can be made by the schedulers, potentially causing resource contention. Mercury Runtime resolves such conflicts as follows:

GUARANTEED - GUARANTEED By design the central scheduler prevents this type of conflicts by linearizing allocations. This is done by allocating a GUARANTEED container only when it is certain that the target node has sufficient resources.

GUARANTEED - QUEUEABLE This occurs when a central scheduler and the distributed scheduler(s) allocate containers on the same node, causing the node’s capacity to be exceeded. Following the semantics of § 3.2, any cross-type conflict is resolved in favor of GUARANTEED containers. In the presence of contention, (potentially all) running QUEUEABLE containers are terminated to make room for any newly arrived GUARANTEED. If GUARANTEED containers are consuming all the node resources, the start of QUEUEABLE ones is delayed until resources become available.

QUEUEABLE - QUEUEABLE This occurs when multiple distributed schedulers allocate containers on the same target node in excess of available resources. Mercury Runtime on the node enqueues the requests (see Figure 3) and thereby prevents conflicts. To improve job-level latency, we explore a notion of priority among QUEUEABLE containers in § 4.

When a QUEUEABLE container is killed there is potentially wasted computation. To avoid this, Mercury supports promoting a running QUEUEABLE container to a GUARANTEED one. A Job Manager can submit a promotion request to the Mercury Runtime, which forwards it to the central scheduler for validation. The promotion request will succeed only if the central scheduler determines that the scheduling invariants would not be violated.

4 Framework Policies

In the previous section we presented our architecture and the lifecycle of a resource request. We now turn to the policies that govern all scheduling decisions in our system. For ease of exposition we group the policies in three groups: Invariants enforcement, Placement, and Load shaping, as described in the following subsections.
4.1 Invariants Enforcement Policies

These policies describe how scheduling invariants are enforced throughout the system.

Invariants for Guaranteed containers Supporting scheduling invariants for centralized scheduler designs is well studied [1, 2, 14]. Furthermore, widely deployed Hadoop/YARN frameworks contain robust implementations of cluster sharing policies based on capacity [1] and fairness [2]. Hence, Mercury’s central scheduler leverages this work, and can enforce any of these policies when allocating Guaranteed containers.

Enforcing quotas for Queueable containers The enforcement of invariants for distributed schedulers is inherently more complex. Recall that applications have very limited expectations when it comes to Queueable containers. However, cluster operators need to enforce invariants nonetheless to prevent abuses. We focus on one important class of invariants: application-level quotas. Our Mercury Runtime currently provides operators with two options: (1) an absolute limit on the number of concurrently running Queueable containers for each application (e.g., a job can have at most 100 outstanding Queueable containers), and (2) a limit relative to the number of Guaranteed containers provided by the central scheduler (e.g., a job can have Queueable containers up to $2 \times$ the number of Guaranteed containers).

4.2 Placement Policies

These policies determine how requests are mapped to available resources by our scheduling framework.

Placement of Guaranteed containers Again, for central scheduling we leverage existing solutions [1, 2]. The central scheduler allocates a Guaranteed container on a node, if and only if that node has sufficient resources to meet the container’s demands. By tracking when Guaranteed containers are allocated/released on a per-node basis, the scheduler can accurately determine cluster-wide resource availability. This allows the central scheduler to suitably delay allocations until resources become available. Furthermore, the scheduler may also delay allocations to enforce capacity/fairness invariants.

Distributed placement of Queueable containers Our objective when initially placing Queueable containers is to minimize their queueing delay. This is dependent on two factors. First, the head-of-line blocking at a node is estimated based on: (1) the cumulative execution times for Queueable containers that are currently enqueued (denoted $T_q$), (2) the remaining estimated execution time for running containers (denoted $T_r$). To enable this estimation, individual Job Managers provide task run-time estimates when submitting containers for execution. Second, we use the elapsed time since a Queueable container was last executed successfully on a node, denoted $T_i$, as a broad indicator of resource availability for Queueable containers on that node. The Mercury Runtime determines at regular intervals the ranking order $R$ of a node as follows:

$$ R = T_q + T_r + T_i $$

Then it pushes this information to the Mercury Coordinator that disseminates it to the whole cluster through the heartbeat mechanism. Subsequently, each distributed scheduler uses this information for load balancing purposes during container placement. We build around a pseudo-random approach in which a distributed scheduler allocates containers by arbitrarily choosing amongst the “top-$k$” nodes that have minimal queuing delays, while respecting locality constraints.

4.3 Load Shaping Policies

Finally, we discuss key policies related to maximizing cluster efficiency. We proceed from dynamically (re)-balancing load across nodes, to imposing an execution order to Queueable containers, to node resource policing.

Dynamically (re)-balancing load across nodes To account for occasionally poor placement choices for Queueable containers, we perform load shedding. This has the effect of dynamically re-balancing the queues across machines. We do so in a lightweight manner using the Mercury Coordinator. In particular, while aggregating the queuing time estimates published by the per-node Mercury Runtime, the Coordinator constructs a distribution to find a targeted maximal value. It then disseminates this value to the Mercury Runtime running on individual machines. Subsequently, using this information, the Mercury Runtime on a node whose queuing time estimate is above the threshold, selectively discards Queueable containers to meet this maximal value. This forces the associated individual Job Managers to requeue those containers elsewhere.

Observe that these policies rely on the task execution estimates provided by the users. Interestingly, even in case of inaccurate estimates, re-balancing policies will restore the load balance in the system. Malicious users that purposely and systematically provide wrong estimates are out of the scope of this paper, although our system design allows us to detect such users.

Queue reordering Reordering policies are responsible for imposing an execution order to the queued tasks. Various such policies can be conceived. In Mercury, we are

---

4 Such estimates are currently provided by the users, but can also be derived from previous job executions, and/or be dynamically adjusted as parts of a job get executed.

5 Other methods, such as work stealing, can also be applied. We use load shedding as it naturally fits into a YARN-based implementation.
currently ordering tasks based on the submission time of the job they belong to. Thus, tasks belonging to jobs submitted earlier in the system will be executed first. This policy improves job tail latency, allowing jobs to finish faster. This in turn allows more jobs to be admitted in the system, leading to higher task throughput, as we also show experimentally in § 7.2.3.

**Resource policing: minimizing killing** To minimize preemption/killing of running QUEUEABLE containers, the Mercury Runtime has to determine when resources can be used for opportunistic execution. In doing so, it maximizes the chances of a QUEUEABLE container actually running to completion. We develop a simple policy that leverages historical information about aggregate cluster utilization to identify such opportunities. Based on current and expected future workload, the Mercury Coordinator notifies the per-node Mercury Runtimes regarding the amount of local resources that will be required for running GUARANTEED containers over a given time window. Subsequently, the Mercury Runtime can opportunistically use the remaining resources in that period for QUEUEABLE containers and thereby minimize preemption.

### 5 Application-level Policies

As explained in § 3.1, Mercury exposes the API for applications to request both GUARANTEED and QUEUEABLE containers. To take advantage of this flexibility, each Job Manager should implement an application policy that determines the desired type of container for each task. These policies allow users to tune their scheduling needs, going all the way from fully centralized scheduling to fully distributed (and any combination in between).

In this paper, we introduce the following flexible policy, while we discuss more sophisticated options in our technical report [18].

**hybrid-GQ** is a policy that takes two parameters: a task duration threshold \(t_d\), and a percentage of QUEUEABLE containers \(p_q\). QUEUEABLE containers are requested for tasks with expected duration smaller than \(t_d\), in \(p_q\) percent of the cases. All remaining tasks use GUARANTEED containers. In busy clusters, jobs’ resource starvation is avoided by setting \(p_q\) to values below 100%. Note that fully centralized scheduling corresponds to setting \(t_d = 0\), and fully distributed scheduling corresponds to setting \(t_d = \infty\) and \(p_q = 100\%\). We refer to these policies as only-G and only-Q, respectively.

### 6 Mercury Implementation

We implemented Mercury by extending Apache Hadoop YARN [3]. We provide a brief overview of YARN before detailing the modifications that support our model.

---

**Figure 4: Mercury implementation**

---

**6.1 YARN Overview**

Hadoop YARN [27] is a cluster resource management framework that presents a generalized job scheduling interface for running applications on a shared cluster. It is based on a centralized scheduling architecture, consisting of the following three key components.

**ResourceManager (RM)**: This is a central component that handles arbitration of cluster resources amongst jobs. The RM contains a pluggable scheduler module with a few implementations [1, 2]. Based on the sharing policies, the RM allocates containers to jobs. Each allocation includes a token that certifies its authenticity.

**NodeManager (NM)**: This is a per-node daemon that spawns processes locally for executing containers and periodically heartbeats the RM for liveness and for notifying it of container completions. The NM validates the token offered with the container.

**ApplicationMaster (AM)**: This is a per-job component that orchestrates the application workflow. It corresponds to the Job Manager we use throughout the paper.

### 6.2 Mercury Extensions to YARN

We now turn to the implementation of Mercury in YARN. Further details can be found in JIRA (the Apache Hadoop feature and bug tracking system) [6].

**Adding container types** We introduce our notion of container type as a backward-compatible change to the allocation protocol. The semantics of the containers allocated by the YARN RM match GUARANTEED containers. Hence, as shown in Fig. 4, the YARN RM corresponds to the central scheduler of our Mercury design. QUEUEABLE containers are allocated by an ex novo distributed scheduler component, which we added to the NM.

**Interposing Mercury Runtime** We have implemented Mercury Runtime as a module inside the YARN NM (see Fig. 4) and thereby simplified its deployment. As part of our implementation, a key architectural change we made to YARN is that the Mercury Runtime is introduced as a
layer of indirection with two objectives. First, the Mercury Runtime proxies container allocation requests between an AM and Mercury’s schedulers, thereby controlling how requests are satisfied. This proxying is effected by rewriting configuration variables and does not require modifications to AM. Second, for enforcing execution semantics, the Mercury Runtime intercepts an AM submitted container request to the NM and handles them appropriately. We elaborate on these next.

The AM annotates each request with the weakest guarantee it will accept, then forwards the request using the allocate() call in Fig. 4. Mercury directs requests for Guaranteed resources to the central RM, but it may service Queueable requests using the instance of Mercury’s distributed scheduler running in the NM. When this happens, since it is essentially a process context switch, the Queueable containers (and tokens) for any node in the cluster are issued with millisecond latency. The authenticity of the container allocations made by a distributed scheduler is validated at the target NM using the same token checking algorithm that YARN uses for verifying Guaranteed containers.

To enforce the guarantees provided by the respective container types, Mercury intercepts container creation commands at the NM. As illustrated in Fig. 4, a startContainer() call will be directed to the Mercury Runtime module running in the NM. This module implements the policies described in §4; based on the container type, the Mercury Runtime will enqueue, kill and create containers.

6.3 Distributed Scheduler(s)

The distributed scheduler is implemented as a module running in each NM. We discuss the changes necessary for enforcing the framework policies described in §4.

**Placement** To direct Queueable containers to fallow nodes, Mercury uses estimates of queuing delay as described in §4.2. For computing this delay, the Mercury Runtime requires computational time estimates for each enqueued container. We modified the Hadoop MapReduce [3] and Tez [4] AMs to provide estimates based on static job information. Furthermore, in our implementation, the AMs continuously refine estimates at runtime based on completed container durations. The Mercury Coordinator is implemented as a module inside the YARN RM (Fig. 4). It collects and propagates queuing delays as well as the “top-k” information by suitably piggybacking on the RM/NM heartbeats.

**Dynamic load balancing** Our implementation leverages the Mercury Coordinator for dynamic load balancing. We modified the YARN RM to aggregate information about the estimated queuing delays, compute outliers (i.e., nodes whose queuing delays are significantly higher than average), and disseminate cluster-wide the targeted queuing delay that individual nodes should converge to. We added this information to YARN protocols and exchange it as part of the RM/NM heartbeats. Upon receiving this information, the Mercury Runtime on an outlier node discards an appropriate number of queued containers so as to fit the target. Containers dropped by a Mercury Runtime instance are marked as KILLED by the framework. The signal propagates as a YARN event to the Mercury Runtime, which proxies it to the AM. The AM will forge a new request, which will be requeued at a less-loaded node.

**Quotas** To prevent Queueable traffic from overwhelming the cluster, Mercury imposes operator-configured quotas on a per-AM basis. A distributed scheduler maintains an accurate count by observing allocations and container start/stop/kill events.

7 Experimental Evaluation

We deployed our YARN-based Mercury implementation on a 256-node cluster and used it to drive our experimental evaluation. §7.1 provides the details of our setup. In §7.2, we present results from a set of micro-experiments using short tasks. Then in §7.3, we describe results for a synthetic workload involving tasks with a range of execution times. Finally, in §7.4, we give results from workloads based on Microsoft’s production clusters.

Our key results are:

1. Our policies can translate task throughput gains into improved job latency for 80% of jobs, and 36.3% higher job throughput (§7.2.1).
2. Careful resource policing reduces the preemption of Queueable containers by up to 63% (§7.2.3).
3. On production-derived workloads, Mercury achieves 35% task throughput gain over Stock YARN (§7.4).

7.1 Experimental Setup

We use a cluster of approximately 256 machines, grouped in racks of at most 40 machines. Each machine has two 8-core Intel Xeon E5-2660 processors with hyper-threading enabled (32 virtual cores), 128 GB of RAM, and 10 x 3-TB data drives configured as a JBOD. The connectivity between any two machines within a rack is 10 Gbps while across racks is 6 Gbps.

We deploy Hadoop/YARN 2.4.1 with our Mercury extensions for managing the cluster’s computing resources amongst jobs. We set the heartbeat frequency to 3 sec, which is also the value used in production clusters at Yahoo!, as reported in [27]. For storing job input/output we use HDFS [7] with 3x data replication. We use Gridmix [8], an open-source benchmark that uses workload
traces for generating synthetic jobs for Hadoop clusters. We use Tez 0.4.1 [4] as the execution framework for running these jobs.

**Metrics reported** In all experiments we measure task throughput, job throughput, and job latency for runs of 30 mins. Due to space limitations, we report only task throughput in some cases, however, the full set of results can be found in [18]. Note that for the task and job throughput we are using box plots (e.g., see Fig. 5), in which the lower part of the main box represents the 25-percentile, the upper part the 75-percentile, and the red line the median. Moreover, the lower whisker is the 5-percentile, the upper the 95-percentile, and the green bullet the mean.

### 7.2 Microbenchmarks

In this section we perform a set of micro-experiments that show how Mercury can translate task throughput gains into job throughput/latency gains. For a given workload, we first study how the maximum number of jobs allowed to run concurrently in the cluster affects performance (§ 7.2.1). Then, we experimentally assess various framework policies (as discussed in § 4), including placement (§ 7.2.2) and load shaping policies (§ 7.2.3).

For all experiments of this section we use Gridmix to generate jobs with 200 tasks/job, in which each task executes, on average, for a 1.2 sec duration. We use the only-G and only-Q policies (§ 5).

#### 7.2.1 Varying Number of Concurrent Jobs

In this experiment, we investigate the performance of the system by altering the number of jobs that the scheduling framework allows to run concurrently. For distributed scheduling (only-Q), we set this limit to 100, 150 and 200 jobs. This is compared with the central scheduler (only-G) that implements its own admission control [27], dynamically adjusting the number of running jobs based on the cluster load. Fig. 5 shows that only-Q dominates across the board, and that, given our cluster configuration, 150 concurrent jobs yield the maximum increase of task throughput, i.e., 38% over only-G. This task throughput improvement translates to improvement in both job throughput and latency (higher by 36% and 30%, respectively, when compared to only-G). Low job limits (100 jobs) fail to fully utilize cluster resources, while high limits (200 jobs) impact latency negatively.

In the following experiments, we use the 150-job limit, as this gives the best compromise between job throughput and latency, and explore other parameters. At each experiment we adjust the job submission rate, so as to have sufficient jobs at each moment to reach the job limit.

#### 7.2.2 Placement Policies (Varying Top-\(k\))

As discussed in § 4.2, whenever a distributed scheduler needs to place a task on a node, it picks among the \(k\) nodes with the smallest estimated queuing delay. Here we experiment with different values for \(k\). Our results are shown in Fig. 6. The biggest gains are achieved for \(k=50\), with 44.5% higher task throughput compared to only-G. Lower values (\(k=20\)) leave nodes under-utilized, while higher values (\(k=100\)) place tasks to already highly-loaded nodes. In both cases, higher load imbalance is created, leading to lower task throughput. Therefore, in the remainder of the experiments we use \(k=50\).

#### 7.2.3 Load Shaping Policies

In this section we study the load shaping policies that were presented in § 4.3.

**Balancing node load and queue reordering** We experiment with different ways of rebalancing node queues. We synthetically cause imbalance by introducing few straggler nodes that underestimate queuing delay. Our results are given in Fig. 7. Among the presented policies, (1) only-Q is a basic approach with no rebalancing; (2) only-Q/avg+\(\sigma\) triggers rebalancing actions for any node with a queuing delay which is over mean plus one standard deviation (\(\sigma\));
Figure 6: Task throughput for varying top-\(k\).

Figure 7: Task throughput and job latency for various load balancing policies.

Figure 8: Desired and actual maximum percentage of memory given to QUEUEABLE containers at each node.

<table>
<thead>
<tr>
<th>Memory limit per node for QUEUEABLE containers</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean containers killed / node</td>
<td>287</td>
<td>428</td>
<td>536</td>
<td>780</td>
</tr>
<tr>
<td>Mean slot utilization for QUEUEABLE</td>
<td>11.4%</td>
<td>16.7%</td>
<td>20.9%</td>
<td>28.4%</td>
</tr>
</tbody>
</table>

Table 1: Effectiveness of maximum memory limit for QUEUEABLE containers.

(3) only-Q/avg+2\(\sigma\) is as above with 2 standard deviations; (4) only-Q/avg+2\(\sigma\)/reorder is as above with reordering of containers in the queue (favoring jobs submitted earlier). Imbalances limit the task throughput gains of only-Q to 8.5% over our baseline only-Q. Subsequent refinements improve the resulting gains by up to 39.8%. Note that reordering reduces average job latency: as jobs exit the system, new jobs start, and by imposing fresh demand for resources drive utilization higher. We measured frequency of task dequeuing to be at an acceptable 14% of all tasks.

Resource policing: minimizing container killing To show how resource policing (discussed in § 4.3) can be used to minimize container killing, we create a GridMix workload that generates a stable load of 70% using GUARANTEED containers, that is, 30% of the slots can be used at each moment for QUEUEABLE containers. At the same time, we are submitting QUEUEABLE containers and observe the average number of such containers killed (due to the GUARANTEED ones). We set the allowed memory limit for QUEUEABLE containers to 20, 30, 40, and 100% (the latter corresponds to no limit). Our results are shown in Table 1. We also report the average utilization due to QUEUEABLE containers. Our implementation is able to opportunistically use resources leading to utilization gains. However, given a steady compute demand, aggressively utilizing those resources without knowing future demand does cause an increase in task kills.

To address this issue we develop a novel policy using historical cluster utilization data to determine the compute demand for current and future workload due to GUARANTEED containers. Any remaining resources can be used for executing QUEUEABLE containers. We input this information to the Mercury Coordinator, which periodically propagates it to the Mercury Runtime on individual machines. This allows the Mercury Runtime on each node to determine how many of the unallocated resources can be used opportunistically. Fig. 8 shows the actual (dashed line) and the observed (solid line) resources used at a node for executing QUEUEABLE containers. The two lines track closely, demonstrating that our implementation adapts to changing cluster conditions. This also shows that there is no need for strict partitioning of resources.

7.3 Impact of Task Duration

We now explore the impact of task duration, by using task run-times of 2, 10, and 20 sec. We compare only-G and only-Q, parameterized at best, given our § 7.2 experiments. Our results are shown in Fig. 9. As expected, the longer the task duration, the smaller the benefit from using distributed scheduling. In particular, when compared to the centralized scheduling, we get approx. 40% gains both in task and job throughput for jobs with 2 sec tasks. This gain drops to about 14% for jobs with 20 sec tasks. Likewise, average job latency for distributed scheduling is comparable with centralized for 2 sec tasks, but is 60% worse for 20 sec tasks.

Note that for short tasks, to fully utilize the cluster resources, more jobs are admitted in the cluster. For dis-
tributed scheduling, this leads to queuing at the same
time tasks belonging to a bigger number of jobs, which
increases the variance of job duration and thus of job
throughput. We are investigating more involved queue
reordering techniques to further mitigate this issue.

7.4 Microsoft-based Hybrid Workload

Finally we assess our system against a complex scenario.
We create a Gridmix workload that follows the task du-
aration characteristics observed in Microsoft production
clusters, as shown in Fig. 1.

We explore several configurations for our hybrid-QQ
policy (§ 5). Besides only-G and only-Q, we have:

- 50%-Q: all tasks have a 50% chance of being QUEUEABLE
  ($t_D = \infty$, $p_q = 50\%$);
- <5sec-Q: all tasks shorter than 5 seconds are
  QUEUEABLE ($t_D = 5sec$, $p_q = 100\%$);
- <10sec-70%-Q: 70% of tasks shorter than 10 seconds
  are QUEUEABLE ($t_D = 10sec$, $p_q = 70\%$);

In Fig. 10 we report on the task throughput, as well as
the job latency for jobs with various task durations from
this workload. In this mixed scenario, using only-Q
18
17
16
15
12
11
10
9
8
7
6
5
4
3
2
1
0

Figure 9: Task throughput, job throughput and job latency for jobs with increasing task duration.

the execution of short tasks using QUEUEABLE contain-
ers between the execution of long running tasks that use
GUARANTEED ones.

We also provide results for an additional hybrid work-
load in [18].

8 Related Work

Mercury relates to several proposed resource manage-
ment frameworks, which we discuss in this section.

Centralized Cluster resource management frameworks,
such as YARN [27], Mesos [16], Omega [24] and
Borg [28], are based on a centralized approach. We
implemented Mercury as extension to YARN and
experimentally demonstrated performance gains of a
hybrid approach. Borg is similar to YARN in that it
uses a logically centralized component for both resource
management and scheduling. On the other hand, Mesos
and Omega are geared towards supporting diverse,
independent scheduling frameworks on a single shared
cluster. They use a two-level scheduling model where
each framework (e.g., MPI, MapReduce) pulls resources
from a central resource manager, and coordinates multi-
tenant job execution over these resources in an idiom
isolated to that framework. Omega uses an optimistic
concurrency control model for updating shared cluster
state about resource allocation. This model works well
for clusters that retain their resources for a reasonably
long duration; a scheduling framework will almost
always obtain the set of nodes it needs, retries are rare,
and frameworks reach quick consensus on allocations.
In contrast, our approach of dynamic load balancing
works well even for heterogeneous workloads that share
resources at finer granularity.

A central scheduler can reason globally about soft
constraints such as data locality [17, 31], or hard con-
straints including multi-resource sharing [14], capacity
guarantees [1] or fairness [2]. With knowledge of the
workload, a central scheduler can also reason about al-
locations over time to effect reservation-based schedul-
ing [11] and packing [15]. We leverage this rich body
of work for Mercury’s central scheduler. Quasar [13] accounts for both resource heterogeneity and interference during task placement, leading to near-optimal scheduling for long jobs but impacting the latency of short jobs.

HPC schedulers (e.g., SLURM [30], TORQUE [26]) are also centralized job scheduling frameworks that support at most a few hundred concurrent running jobs/sec, orders of magnitude lower than what Mercury targets.

Distributed Systems such as Apollo [9], are built using a fully decentralized approach. These schedulers achieve extreme scalability for low-latency allocations by allowing and correcting allocation conflicts. Lacking a chokepoint for throttling or coordinated feedback, fully distributed techniques maintain their invariants on an eventual manner. Worker nodes in distributed architectures maintain a queue of tasks to minimize time the node spends idle and to throttle polling. Similar to Mercury, Apollo estimates wait times at each node and lazily propagates updates to schedulers. In particular, Apollo uses a principled approach that combines optimizer statistics and observed execution behavior to refine task runtime estimates. These techniques can be incorporated by YARN AMs, which can in turn improve Mercury’s placement and load balancing policies. Note that, unlike Mercury, the scheduler in Apollo is part of the SCOPE [34] application runtime, so operator policies are not enforced, updated, or deployed by the platform.

Executor model Single-framework distributed schedulers focus on a different class of workloads. Sparrow [22] and Impala [19] schedule tasks in long-running daemons, targeting sub-second latencies. This pattern is also used in YARN deployments, as applications will retain resources to amortize allocation costs [4, 29, 33] or retain data across queries [20, 32]. In contrast, Mercury not only mixes heterogeneous workloads with fine granularity, but its API also enables jobs to suitably choose a combination of guaranteed and opportunistic resources.

Performance enhancement techniques Corrective mechanisms for distributed placement of tasks are essentially designed to mitigate tail latency [12]. Sparrow uses batch sampling and late binding [22], which are demonstrably effective for sub-second queries. Apollo [9] elects to rebalance work by cloning tasks (i.e., duplicate execution), rather than shedding work from longer queues. Resources spent on duplicate work adversely affect cluster goodput and contribute to other tasks’ latency. Instead, Mercury uses dynamic load shedding as its corrective mechanism.

Several Big Data schedulers have dynamically adjusted node allocations to relieve bottlenecks and improve throughput [23, 25], but the monitoring is trained on single frameworks and coordinated centrally. Principled oversubscription is another technique often applied to cluster workloads [10] with mixed SLOs. Our current approach with Mercury is intentionally conservative (i.e., no oversubscription) and already demonstrates substantial gains. We can further improve on these gains by enhancing Mercury to judiciously overcommit resources for opportunistic execution.

9 Conclusion

Resource management for large clusters and diverse application workloads is inherently hard. Recent work has addressed subsets of the problem, such as focusing on central enforcement of strict invariants, or increased efficiency through distributed scheduling. Analysis of modern cluster workloads shows that they are not fully served by either approach. In this paper, we present Mercury, a hybrid solution that resolves the inherent dichotomy of centralized-distributed scheduling.

Mercury exposes the trade-off between execution guarantees and scheduling efficiency to applications through a rich resource management API. We demonstrate experimentally how this design allows us to achieve task throughput improvements, while providing strong guarantees to applications that need them. The task throughput gains are then translated to job level performance wins by well tuned policies.

The key architectural shift we introduce, has far greater generality than we discussed in this paper. In particular, the Mercury Runtime provides a level of indirection that is being leveraged to scale YARN clusters to over 50K machines by federating multiple smaller clusters [5].
References


