# Reservation-based Scheduling: If You're Late Don't Blame Us!

Carlo Curino[m] Djellel E. Difallah[u] Chris Douglas[m]
Raghu Ramakrishnan[m] Sriram Rao[m]

Microsoft Corp. Technical Report: MSR-TR-2013-108

[m] : {ccurino, cdoug, raghu, sriramra}@microsoft.com, [u]: djelleleddine.difallah@unifr.ch

**Abstract**

In this paper, we tackle the problem of running a rich mix of jobs, including job pipelines with gang-scheduling and deadlines, by carefully separating the following concerns: (1) determining resource requirements for a job/pipeline (taking deadlines and other considerations into account), and (2) ensuring predictable allocation of requested resources.

We propose a *resource description language* that allows each job to specify its resource needs abstractly to the system, exposing many alternative ways of satisfying the job's resource needs. This gives the system flexibility in allocating resources across several jobs, while also allowing it to plan ahead and determine whether it can satisfy any given job's resource request. We show the power of this approach by presenting a scheduling framework that uses these rich resource-requests to ensure *predictable resource allocation for production jobs while minimizing latency for best-effort jobs*. Our framework relies on admission control (and quick adaptation to changes in cluster usage) to ensure predictable SLAs for resource reservations, and uses work-preserving preemption to dynamically reallocate resources.

We demonstrate these techniques by building *Rayon* as extension to YARN (Hadoop 2.x). This allows us to validate our work in a real context and against some of the most popular schedulers. Our experimental evaluation is based on micro-benchmarks and ten big-data workloads derived from real-world traces from clusters of Cloudera customers, Facebook, Microsoft, and Yahoo!. We also present the results of running our system at thousands of jobs an hour on a large 25,000 slots cluster.



*CONF 'yy*, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

## 1. Introduction

Over the past decade, scale-out computing over large clusters (with thousands of nodes) has become ubiquitous, following its success in web companies such as Facebook, Google, LinkedIn, Microsoft, Quantcast, and Yahoo!. By centralizing data storage and providing massive computational resources to analyze the data, these clusters gave rise to "big-data analysis" and opened the door to data analytics as a cloud service.

The types of analysis people run have diversified from MapReduce jobs to SQL, interactive analytics, stream processing, iterative machine learning, and MPI-style computations; these clusters are now used for everything from research and testing to running production pipelines. This explosion in the complexity and variety of big-data applications has fueled a shift from single-purpose clusters to clusters running a mix of applications including batch jobs, latency-sensitive queries, pipelines with gang-scheduling and deadline requirements, and long running services. In addition to production jobs, which have associated service-level agreements (SLAs), workloads also include best-effort jobs that seek to utilize any spare capacity.

Together with more powerful machines (32 cores, 128GB of RAM, 10-20 drives and 10Gbps NICs are now common), this evolution of cluster workloads is also driving consolidation of dedicated, single-purpose clusters into shared, multi-purpose clusters. In turn, this is motivating extensive redesign of major cluster frameworks such as Hadoop YARN [4], Corona [1], Omega [21], and Mesos [13], with the goal of improving cluster utilization and job metrics. Figure 1 highlights this evolution in the context of the Hadoop ecosystem. Hadoop 1.x was solely geared towards MapReduce computations. In contrast, YARN (i.e., Hadoop 2.x) provides a more general programming surface on which diverse specialized applications can be built.

While differing substantially, all of the above mentioned frameworks introduce a separation between a component handling resource arbitration among jobs, which we will call a *resource manager*, and (one or more) component(s) handling the application workflow, which we will call a *job manager*. An important issue to consider is the division

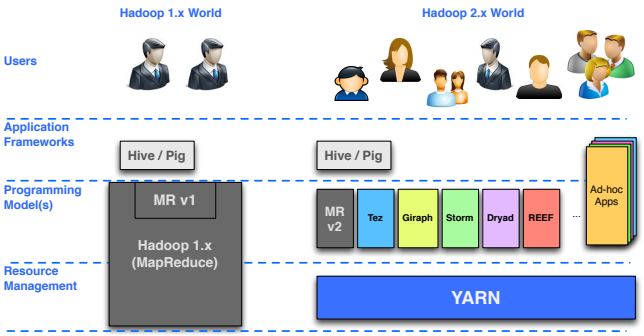


**Figure 1.** Hadoop 1.x vs Hadoop 2.0/YARN

of responsibilities across the resource manager and the job manager, especially in light of the increasingly complex job mixes that cluster frameworks are being asked to handle.

The main goal of this paper is to design a framework to cope with these increasingly complex workloads, especially in the setting of *data analytics as a cloud service*, where a large, shared, multi-tenanted cluster "in the cloud" runs jobs on behalf of many tenants/users. Specifically, we focus on: (1) how to allow jobs to specify their resource needs up front, together with any constraints and any available flexibility, and (2) how to provide *predictable resource allocation* for a rich class of resource-requests.

### 1.1 Motivation

Systems like YARN, Mesos, Omega and Corona focus on job throughput and use simple instantaneous notions of capacity/fairness for resource sharing (e.g., see [4, 19, 21]). Consequently, applications see substantial variability in *when* their resource demands will be met, making it difficult to manage jobs with deadlines or dependencies. Cluster operators today cope with this with very limited tools such as job priorities, queue hierarchies, cluster over-provisioning, and constant manual intervention.

Furthermore, jobs whose scheduling requirements are not explicitly supported by the scheduler are forced to implement ad-hoc measures that affect overall cluster utilization. Two important examples from this class are jobs with gang-semantics, and pipelines (workflows with multiple jobs and time-dependent resource needs).

In order to run a job whose tasks must be gang-scheduled in a system that allocates resources as and when they become available (i.e., trickling the task allocations), the job manager is forced to "hoard" resources until *all* computational tasks have been provided by the scheduler—this is the approach taken in Apache Giraph [2]. We highlight in Figure 2 three key shortcomings of this approach by running Giraph in isolation and together with a sort job: 1) increased runtime for the gang jobs (80% in our experiment), 2) wasted resources during the hoarding phase (the Giraph jobs starts acquiring resources at time 1000 but can only start computing at time 1650, 28% of its total slot allocation is thus wasted), and

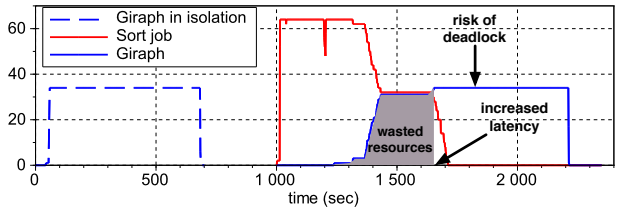


**Figure 2.** Shortcoming in scheduling gang jobs today.

3) risk of deadlocks among multiple gang jobs (not shown). These effects can become much worse in busy clusters.

When running pipelines of jobs, users are forced to statically provision for peak demand, or gamble on completion times despite the fact that pipelines are typically characterized by stringent SLAs (where violation can have substantial economic impact), have a periodic nature [10], and very pronounced differences in resource demands over time (over 10X is common). We confirm our intuitions inspecting several pipelines from Microsoft production grids, we visualize a few in Figure 3.

This state of affairs is taxing for large organizations, and unaffordable for smaller ones. It is also highly unsatisfactory for use in a public-cloud shared service setting, where scale and the contractual nature of the relationship between users and operators exacerbate the problems.

This exposes the need for a richer vocabulary to express job requirements. Key extensions will include explicit management of time, gang, and inter-job dependencies. Next we propose a framework capable of providing such expressivity boost.

### 1.2 Proposed Approach

Our approach decomposes the hard problem of managing complex workloads of jobs with completion deadlines, gang-scheduling, and pipelines into two sub-problems: 1) *resource definition*: i.e., determining the amount of resources required by the job, and 2) *predictable resource allocation*: i.e., the problem of determining whether a job can be admitted, when it should be run, and with how many resources.

The first problem is mainly dependent on the given job (e.g., its parallelism, code efficiency, data skew), while the second is a scheduling/planning problem that requires consideration of other jobs and their needs; in this paper we focus on the latter[1].

We argue that the *job manager* should focus only on translating job-specific requirements (such as deadlines, latencies, cost-budgets, etc.) into appropriate resource requests, while the *resource manager* should focus on delivering predictable resource allocations over time, by carefully planning resource assignments and dynamically arbitrating instantaneous resource requests from running jobs. This sep-

[1] Assuming that resource definitions are given is reasonable since production jobs are typically periodic [10], and thus amenable to history-based resource prediction [20].

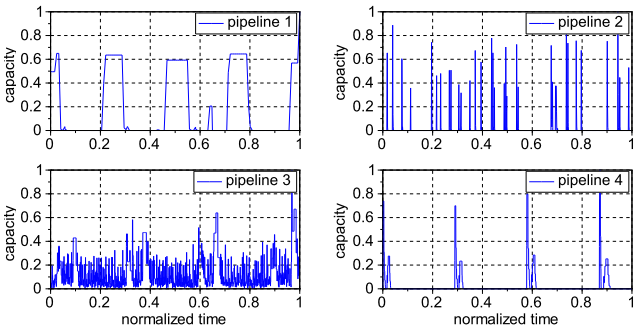


**Figure 3.** Production pipelines from Microsoft clusters, normalized time and capacity.

aration of concerns requires that the resource manager be able to handle an expressive class of resource requests.

To this purpose we introduce an extensible *Resource Definition Language (RDL)* in terms of two axes: *space*: type and amount of resources required, and constraints on the allocation such as gang semantics, degree of parallelism, and locality preferences; and *time*: the window of time within which a request must be satisfied, the duration of the request, and any temporal constraints (e.g., inter-job dependencies in job pipelines). The latter dimension, in particular, is not addressed satisfactorily by current systems (see Section 1.1), and represent a key extension we propose in this paper.

The job manager relies upon the resource manager to reliably satisfy resource requests expressed in RDL, including constraints related to execution time windows and job dependencies. This is a much more flexible building block for meeting SLAs in complex workloads than current systems offer. At the same time, the RDL resource request exposes as much scheduling flexibility as possible to the resource manager, in sharp contrast to current state-of-the-art systems, where only instantaneous resource needs are exposed. This opens up opportunities to improve system-wide utilization.

Our design of the resource manager consists of two parts: the *planner*, which leverages pluggable policies to organize the cluster's agenda for the next time window (i.e., which jobs are to be run, when, and with what resources), and the *allocator*, which exploits work-preserving preemption (see [3, 4, 9]) to elastically grant and revoke access to resources based on the planner's indications.

***Scope and contributions*** This paper makes the following contributions. First, we present an expressive resource description language (RDL) to allow jobs to specify their resource requirements in both space and time dimensions. Second, we present an extensible scheduling framework called *Rayon* that provides predictable resource allocation to job requests specified in RDL. Third, we implement *Rayon* as an open-source extension of the Apache Hadoop YARN framework. Fourth, we provide an extensive experimental evaluation of *Rayon* based on workloads derived from Cloudera, Facebook, Linkedin, Microsoft and Yahoo! production clusters, together with several synthetic micro-benchmarks. Comparing our system against a production-class scheduler we observe: 1) an almost tuning-knob free experience, 2) 15% increase in cluster throughput, 3) 30 to 50% increase in cluster utilization, 3) 0% SLA violations (vs 15% for the baseline) while accepting over 96% of SLA jobs.

There are important related problems in designing optimal placement policies and defining strategy-proof incentives for resource definition; these problems are beyond the scope of this paper. We believe that the framework we propose provides a good substrate for exploring these issues, and will facilitate their practical impact through open-source contributions.

## 2. The *Rayon* Framework

In this section, we define each of the components and interfaces in *Rayon*.

We first describe how *production jobs* are admitted to the system. The job manager estimates the demand generated by a job or pipeline and encodes its constraints as a *request expression* in the *Resource Definition Language* (RDL) defined in Section 2.1. This step is not the focus of our paper; it can be based on semantic understanding of the job, analysis of execution traces from past runs, or some combination of the two [16, 18].

The resource manager maintains an *inventory* of cluster capacity by tracking future reservations. The planner accepts a job if the constraints in the request expression are satisfiable, given the inventory. We describe how the planner makes this determination in Section 2.2.1.

Once a request expression is accepted and accounted for in the inventory, it defines a *contract* for allocating resources. We describe how this contract is enforced in Section 2.2.2.

We also consider the effect of *Rayon* on *best-effort jobs* with valid request expressions i.e., they place no constraints on their reservation and are satisfiable for all inventories. "Idle" resources in the inventory are distributed not only among these jobs, but also among production jobs capable of running before their allocation in the inventory. We describe how *Rayon* uses work-conserving preemption to resolve contention efficiently at runtime in Section 2.2.3.

### 2.1 Resource Description Language

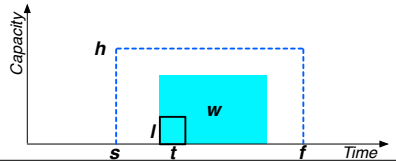


**Figure 4.** Illustration of constraints expressed in RDL

An RDL *request expression* is one of the following types of expressions, as shown in Figure 4.

- An *atomic expression* of the form $\langle l, h, t, w \rangle$, where: (1,2) $l$ / $h$ are the minimum / maximum parallelism for the step; a valid allocation of capacity at a time quanta is either 0 or in the range $[l, h]$. (3) $t$ is minimum duration of consecutive allocations; a valid allocation persists for at least $t$ steps. (4) $w$ is the threshold of work necessary to complete the reservation; the expression is satisfied iff the sum of all valid allocations is exactly $w$.
- A *choice expression* of the form *any*$\{e_1, e_2, \ldots e_n\}$. It is satisfied if *any* one of the expressions $e_i$ is satisfied.
- A *union expression* of the form *all*$\{e_1, e_2, \ldots e_n\}$. It is satisfied if *all* the expressions $e_i$ are satisfied.
- A *dependency expression* of the form *order*$\{e_1, e_2, \ldots e_n\}$. It is satisfied if all expressions $e_i$ can be satisfied strictly before $e_{i+1}$.
- A *window expression* of the form *window*$\{e, s, f\}$, where $e$ is an expression and $[s, f)$ is a time interval. This expression bounds the portion of the inventory in which $e$ can be satisfied. We denote this slice of the inventory as $inv_{[s,f)}$.

It is easy to see that RDL allows job managers to express both *malleable* and *rigid* requirements for *atomic* computation stages of a job. For example, a MapReduce job supporting preemption may select a low $t$ and admit loose constraints on its parallelism; in contrast, an MPI job may define its parallelism inflexibly and require uninterrupted operation until $w$ is met. While best-effort jobs do not participate in the reservation protocol, we can still define them formally as an atomic expression $\langle 1, h, 1, 0 \rangle$. This is trivially satisfied by the admission control and affects no other placements. Note that all request expressions (except $window$) typically admit multiple solutions for a given inventory.

It is straightforward for a job manager to compose complex pipelines and DAGs in RDL using the other operators. In addition to connectors like *all* and *order*, note that the *any* operator allows a job manager to propose multiple physical plans for a single logical plan, granting the planner significant flexibility. The *window* operator can constrain a job to receive resources only during a particular interval.

An expression in RDL is resolved against an inventory of cluster resources, which tracks reservations over an interval $inv_{[s,e)}$. At any point in this interval, $inv[t]$ contains all the reservations promised to jobs at time $t$, which comprise some fraction of the projected capacity at $t$.[2] In the following section, we describe how a job manager might encode its requirements in RDL.

#### 2.1.1 Example: A Recurring MapReduce DAG

Consider an example construction of a request expression $e$ in RDL. For purposes of illustration, we model a recurring workflow of three MapReduce jobs $\{A, B, C\}$ processing input available at time $t_s$ with a deadline of $t_e$. Assume that $C$ depends on the output of both $A$ and $B$. Since our constraints are on the entire pipeline, we place a $window$ expression at the top level, $e = window\{e', t_s, t_e\}$. Because we can run $A$ and $B$ concurrently, but both must complete before $C$ can use its resources, we represent our DAG dependencies as $e' = order\{all\{e_A, e_B\}, e_C\}$, yielding an expression that expresses all our temporal constraints: $window\{order\{all\{e_A, e_B\}, e_C\}\}$.

Again, estimating the runtime of MapReduce jobs is outside our scope and an area of active research. For this example, assume the job manager estimates the number of tasks $h$ and their duration $r$ based on the size of the input data, then uses these to define the total work $w$ for each job. Based on observations of fanout from $A$ and $B$, the job manager can estimate the input to $C$ and consequently its values for $w$ and $h$.

The job manager can use estimated task runtime to define $t$, to ensure that every allocation completes at least some tasks. As we discuss in Section 3, we implement work-conserving preemption for MapReduce tasks. Assuming an estimate of the context switching cost, the job manager can define $t \leq r \leq w$ for each of its reservations, where $t$ is estimated from observed context switching costs. We arrive at our final request:

$$window(order(all(\langle 1, h_A, t_A, w_A \rangle, \langle 1, h_B, t_B, w_B \rangle) \\ \langle 1, h_C, t_C, w_C \rangle\}, t_s, t_e\}$$

While we do not make any completeness claims about the RDL language, we find it sufficient to naturally capture a broad class of practical scenarios.

### 2.2 Resource Manager

The resource manager determines whether a request expression violates any existing contracts and, once accepted, ensures that a job receives its promised allocation. This problem is referred to in literature as *commitment on arrival* [17] and it is handled in *Rayon* by a component called the *planner*. To service latency-sensitive, best-effort jobs, we draw on the pool of resources unclaimed by active contracts.

Our environment is inherently chaotic. Managing a cluster of unreliable machines necessarily exposes the planner to violations caused by mispredictions of its future capacity. While no plan is secure against cluster collapse, we maintain high utilization and satisfy reservations by planning conservatively and aggressively back-filling the cluster with best-effort jobs.

We argue that attempting to tightly control this environment is hopeless and the only reasonable design builds mechanisms that robustly adapt to changing conditions at runtime. In this context, the plan is an objective for our *allocator* and *adapter* components, which leverage preemption to adjust the execution environment to achieve resource-oriented goals.

The rest of this section describes the roles of planner, allocator and adapter at a conceptual level.

[2] If overcommitted, the reserved capacity may exceed projected capacity.

### 2.2.1 Planner

On receipt of a request expression $e$ and preference function $rank$ generated by a job manager, the planner executes the admission control algorithm shown in Algorithm 1.

The algorithm is expressed here as a set of generator functions that enumerate all solutions to $e$. Note that `yield` retains the state of the function when it returns to the caller, so subsequent calls to a node in the parse tree will return the *next* inventory satisfying the expression (if it exists). The $rank$ function is used to select the "best" placement, including whether rejecting the job would be better than a solution that satisfies the placement constraints for $e$. Every call to `accept` is overloaded to resolve a node in the request expression against an inventory.

**Algorithm 1:** Planner (RDL Resolution)

**Input**: Description of resource reservation $e$, inventory $inv$ recording planning interval $inv_{[s,f)}$, transitive function `rank` to order solutions by preference
**Output**: The most-favored inventory considering $e$

```
def resolve(e : RDL, inv : Inv, (rank) : (Inv, Inv) → Inv): Inv
    inv' ← nil
    forall s ∈ accept(e, inv)
        inv' ← rank(inv', s) // select preferred alloc
    rank(inv, inv') // select alloc or existing inventory
def accept((e, s, f) : window, inv : Inv): Inv
    accept(e, inv_[s,f))
def accept({e_1, ..., e_2} : any, inv : Inv): Inv
    forall s ∈ accept(e_1, inv)
        yield s
    accept(any{e_2, ..., e_n}, inv)
def accept({e_1, ..., e_2} : all, inv : Inv): Inv
    forall s ∈ accept(e_1, inv)
        yield accept(all{e_2, ..., e_n}, s)
    nil
def accept({e_1, ..., e_2} : order, inv : Inv): Inv
    forall s ∈ accept(e_1, inv)
        yield accept(order{e_2, ..., e_n}, inv[last(e_1, s), inv.f))
    nil
def accept(⟨l, h, w⟩ : atomic, inv : Inv): Inv
    // yield valid placements in inv
    nil
def last(e : RDL, inv : Inv): Int
    // last allocation to e in inv
```

Algorithm 1 finds a satisfying placement for any arbitrary RDL expression if it exists, but it has dire complexity properties. This is due to the inherent complexity of the underlying problem.

**Theorem 2.1.** *The problem of determining whether N resource requests expressed in RDL can be satisfied is NP-complete.*

A sketch of proof is given by reducing the known NP-complete problem of Job-Shop [11] to our problem.

*Proof.* Given a job-shop problem with N jobs and M machines, we represent each job as an atomic RDL expressions $< 1, 1, w, w >$, where $w$ is the job duration, and try to assign all jobs in an inventory of capacity M and time-horizon $t$. If an allocation is (not) found we (increase) decrease $t$ until we find the smallest inventory that fits all the jobs. This is the solution of the original NP-problem. Since it took us polynomially many applications of our problem to solve a known NP problem, our problem must also be NP. □

This completeness result, prior impossibility results [17] on designing optimal algorithms for commitment on arrival scheduling problems, and the practical considerations about fast-evolving conditions in large clusters justify us in focusing on robust heuristics for our YARN-based implementation. In particular, we explore in Section 3 a series of greedy heuristics that can find solutions in linear time for a practically important subset of RDL expressions.

### 2.2.2 Allocator

The job manager and planner communicate in RDL and generate reservations in a logical plan, but we have yet to allocate resources that accomplish work. A job accepted by the planner has entries in the inventory describing its reservations in a given time interval. When that interval arrives, the allocator enforces all reservations from the inventory.

As shown in Algorithm 2, this is often straightforward. By simply iterating over the plan, we can satisfy all our guarantees in that interval and distribute any "idle" resources according to whatever policy— fairness, capacity, priority, etc.— is important for our deployment. Similarly, when cluster conditions or over-subscription force a violation in this interval, we defer to a policy to determine how to address it. For example, the policy could elect to cancel an iteration of a recurring pipeline if a subsequent stage can handle it.

Frequent changes to these allocations are not disruptive to individual jobs, due to an implementation of work-conserving preemption. We defer the details to Section 3, but a mechanism that allows best-effort jobs to make persistent progress in brief windows of fallow capacity proved essential to maintaining high utilization amid fluctuating resource commitments.

So far, each refinement of our logical plan combines a static resource expression from a job manager to an instance of the inventory; both capture a snapshot of the cluster state that will mutate before and during that job's execution. To accommodate changes to the assumptions that grounded its logical plan, *Rayon* employs dynamic mechanisms we describe in the following section.

### 2.2.3 Adapter

Both the job manager and the resource manager may need to modify the reservations committed to the inventory by the admission control. We refer to both mechanisms as *dynamic adaptation* of the reservation plan.

The original estimates produced by the job manager may be inaccurate due to properties of the data— skew, re-executions induced by data corruption, unanticipated spikes

**Algorithm 2:** Allocator (Scheduling Resources)

**Input**: Inventory $inv$, current time $t$, aggregate capacity $c$, dictionary of jobs $J$, policy funciton $P$
**Output**: Allocation for this time slice

```
def allocate(inv : Inv, t : Int, c : Alloc, J : Job[], (P) : Policy):
Alloc
    a ← ∅
    d_r ← ∑ inv[t]
    if d_r > c then
        // Overcommitted or capacity changed
        // Policy manages violation
        inv[t] ← scale(P, c, inv[t])
    foreach r ∈ inv[t] do
        a ← a ∪ min(r.alloc, J[r].demand)
    // distribute excess capacity using policy
    assign(a, P, J, c − ∑ inv[t])
def assign(a : Alloc, (P) : Policy, J : Job[], c : Alloc): Alloc
    // assign remaining resources to jobs using
    // fairness, capacity, or other metrics
def scale((P) : Policy, c : Alloc, inv : Inv): Inv
    // reassigns committed resources
    // according to policy function
```

in data volume— or to environmental factors— resource contention with other jobs, hardware failures, or cluster faults. In these conditions, the job manager may need to *renegotiate* its reservations with the adapter to achieve its objective.

Symmetrically, the job manager is equally exposed to hardware faults that affect cluster capacity, and it may need to reevaluate its commitments after significant fluctuations. In one frequent case, jobs complete before their reservation is expired, and often before they even enter a reserved interval in the inventory. Evicting these reservations frees resources for newly submitted production work, though it creates a new challenge for the planner: freed reservations create usable, but suboptimal gaps in the inventory. If the adapter were to *replan* the inventory, it could pack these reservations more tightly.

Techniques for complete replanning are expressible in our existing model. If one were to generate a request expression $all\{J_1, \ldots, J_n\}$ of all jobs $J_i$ with reservations in the inventory, Algorithm 1 will find a solution satisfying all outstanding contracts.[3] Given the aforementioned complexity, any replanning will need to make extensive use of heuristics to make such compaction practical.

While this conceptual model is fully general, our current implementation supports only support basic forms dynamic adaptation: for replanning, we implement a continuous updating of the inventory, but we do not reassign all jobs allocations at every step; for renegotiation, our current implementation only allows forfeiting and requesting complete reservations.

[3] Particularly given the flexibility of the *any* operator, this technique requires more deliberate accommodation of existing reservations.

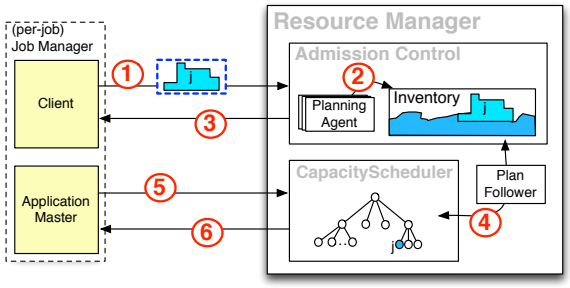


**Figure 5.** YARN-based implementation of *Rayon*.

## 3. Implementing *Rayon*

We use the Apache Hadoop 2.1.0 / YARN [4], an extensive redesign of earlier versions of Hadoop, as the starting point for implementing *Rayon*. We provide an overview of YARN in Section 3.1, and then describe our *Rayon* implementation in Section 3.2.

### 3.1 YARN Overview

YARN consists of the following key components:

- *ResourceManager* (RM): This is a central component which handles arbitration of cluster resources amongst jobs. The RM contains a pluggable scheduler which enforces instantaneous invariants (viz., capacity, fairness, locality) while trying to maximize cluster throughput.
- *NodeManager* (NM); This is a per-node daemon which monitors and polices the local use of resources on behalf of the RM.
- *Client* (Cl): a client-side component used by the user to submit jobs to the RM.
- *ApplicationMaster* (AM): This is a per-job component which is responsible for orchestrating the application workflow. Whenever a user submitted job is accepted by the RM, an AM is the first process started on one of the nodes in the cluster. That is, the RM notifies the NM on the node to spawn the AM. The AM then negotiates with the RM for access to more resources (i.e., containers in YARN jargon).

The YARN RM contains multiple scheduler implementations including the *CapacityScheduler* and *FairScheduler*. The former has received the most thorough practical validation, and it is currently deployed at scale on all of Yahoo!'s grids [22]. We chose this as a our underlying scheduler to build *Rayon*.

The CapacityScheduler enables sharing of cluster resources through a notion of capacity. That is, the cluster resources are exposed to jobs as a set of queues where each queue has access to a minimum guaranteed share of the overall cluster resources, i.e., the queue capacity. Queues are FIFO, can be organized hierarchically, and expose tunables defining their guaranteed, maximum, and per-user shares

### 3.2 Implementing *Rayon* using YARN

Our implementation involves several changes to the YARN RM which collectively enable predictable resource allocation to production jobs. First, we extend the YARN job submission protocol to support explicit resource reservation and implement explicit admission control in the RM (see Section 3.2.1). Second, we modify the CapacityScheduler to be fully dynamic in creating/reconfiguring/deleting queues, and build a Plan Follower component that dynamically updates the CapacityScheduler configuration to mirror the state of the plan (see Section 3.2.2).

#### 3.2.1 The Planner (Admission Control)

Implementing the cluster resource planner module in the YARN RM involves the following set of changes:

1. We extended the YARN client's job submission protocol with a new API to allow production jobs to specify their resource demands to the RM using the RDL described in Section 2. This in turn corresponds to an invocation of our admission control module—step 1 of Figure 5.
2. We implemented a new admission control module in the YARN RM. The admission control handles a job's RDL reservation requests by searching for allocations of the request in the cluster resource inventory—step 2 in Figure 5. If a solution is found that does not violate previously accepted jobs, the job is accepted, and a *reservation handle* is returned to the Client—step 3 of Figure 5. On the other hand, if the RDL expression is not satisfiable by the current inventory, the call fails and the job is rejected. Reservation handles can be used to submit multiple jobs to a single reservation which is common in practice for running pipelines.

In Section 2 we showed that our placement problem is NP-complete. We thus focus on robust heuristics to search for a valid placement of jobs. We call these heuristics (planning) *agents* since they operate on the inventory on behalf of a job.

***Planning Agents*** More precisely, we turn our attention to greedy agents that have runtime linearly proportional to the size of the inventory, and cover important special cases of the RDL language, such as: 1) an individual malleable job with a deadline (e.g., a MapReduce job to be run before 3pm today), 2) a rigid job without a deadline (e.g., a Giraph job to be run as soon as possible), and 3) a pipeline composed only of atomic expressions connected by *order* operators (e.g., Oozie workflows, for which we describe the overall resource skyline). These three cases correspond to three agents we implemented: Fluid, FifoGang, Skyline. All three can be implemented with a single pass over the inventory slots, trying to fit either partial or complete portions (gang semantics of not) of the job reservation in each time slot, and sliding forward or backward (deadline vs latency) if the reservation cannot be fit in this portion of the inventory.

Note that these greedy algorithms might conservatively reject jobs, as they do not consider re-planning of previously accepted jobs. On the other hand, the overhead introduced in admission control is very low. We have experimentally validated this claim, and our current implementation can place 640 jobs, from the SWIM workload [8], every second. This handles the load of even the busiest clusters we studied. Agents are pluggable and represent a key extensibility point for our system. We believe they provide a good substrate for research in this domain.

#### 3.2.2 Implementing The Allocator

The *allocator* of Section 2.2.2 is implemented in YARN by a combination of a Plan Follower and a modified version of the CapacityScheduler.

1. The Plan Follower of Figure 5 runs in a loop, and updates the configuration of the CapacityScheduler by creating a queue for each currently active reservation, and sizing it based on the allocation in the plan. This is step 4 in Figure 5.
2. The AM dynamically petitions the RM for resources, step 5 of Figure 5.
3. The CapacityScheduler in turn allocates/revokes resources to/from the AM based on the queue's capacity (i.e., the queue to which the job was bound to at submission time). This results in step 6 of Figure 5.

In essence, our implementation tries to provide predictable resource allocation to jobs by dynamically reconfiguring the capacity scheduler queues.

The CapacityScheduler has been modified in order to enable dynamic creation/reconfiguration/destruction of queues (normally a limited and heavy-handed process). We experimentally tested how quickly we can update its configuration, and how many concurrent queues we can handle (it was designed for tens of queues, and we use hundreds to thousands). Neither of the two appear to be of any practical concern.

#### 3.2.3 Implementing the Adapter

The adapter is also implemented in the Plan Follower component. Beside tweaking the scheduler configuration, the Plan Followers engage in dynamic replanning by propagating information about job execution progress to the inventory. As an example consider Figure 6, where we show the reservation and actual runtime for a pipeline with three stages. This corresponds to a RDL expression:

$$window\{order\{\langle 1, 10, 240, 2400\rangle, \langle 1, 20, 120, 2400\rangle, \langle 1, 15, 120, 1800\rangle\}, 0, 800\}$$

This experiment ran on a mostly idle cluster. Given this abundance of resources, our system allowed the job to run before its reservation and with more parallel tasks than

initially planned[4]. At time 650 the job completed and the adapter removed its reservations from the inventory.

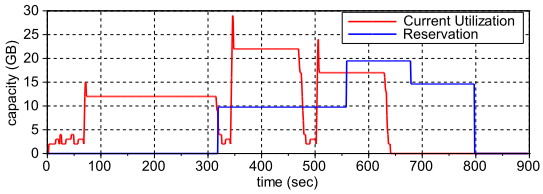


**Figure 6.** Microbenchmarks showing different agents placing skylines and gang jobs, and actual job execution.

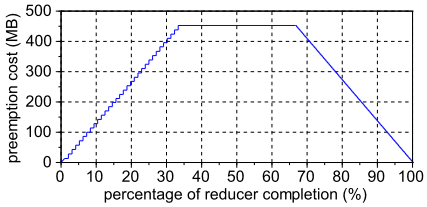


**Figure 7.** Preemption cost as a function of reducer progress.

***Work-conserving preemption*** We conclude our implementation section with few notes regarding the preemption mechanism we exploit in *Rayon*. Work-conserving preemption is an important building block for our system. In this work, we leverage the form of checkpoint-based preemption recently implemented in YARN [4]. This is also comparable with prior work in [5] and parallel efforts in [9]. Key to our discussion is quantifying the cost of preemption. We measure this by preempting the reducers of a 100GB MapReduce sort job at different point in their execution. Figure 7, shows the relationship between task progress and cost of preemption, expressed in terms of the size of the data to be checkpointed. This ranges from near zero (early and late in the execution), up to 450 MB of data in the middle. Given the checkpoint service we provide is HDFS-based this corresponds to 7-10 seconds of preemption time. This is typically an acceptable cost when compared with under-utilizing the cluster or killing tasks. Exploring the tradeoffs between preemption and cluster throughput is beyond the scope of this paper, but clearly of interest to job manager implementations.

## 4. Experimental Evaluation

We deployed our prototype implementation of the *Rayon* framework along with our modifications to the Apache Hadoop YARN version 2.1.0 on a 256-node cluster and used it to drive a two-part experimental evaluation. The first part of the evaluation compares *Rayon* against the stock YARN CapacityScheduler (CS) using previously published (MapReduce) cluster workloads from various companies [7, 8] (Section 4.2), and examines cluster metrics such as cluster utilization and resource allocation SLA violations. The

[4] The job also exposed slightly less parallelism than it could use.

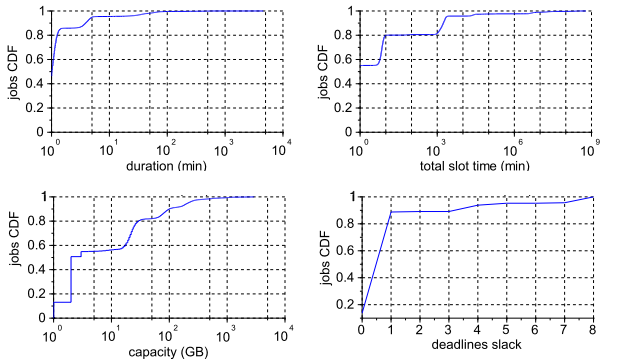


**Figure 8.** Job distribution for duration, total slot time, parallel capacity, and deadline slack.

second part of the evaluation focuses on the predictable resource allocation aspects of *Rayon* (Section 4.3). We use a diverse workload consisting of a rich mix of: (1) MapReduce jobs, (2) Giraph graph computations, which require gang-scheduling, and (3) job pipelines with time-varying resource demands. We also present simulation results to explore parameter sensitivity of our admission control (Section 4.4), and the impact of misconfigured reservations while running *Rayon* (Section 4.5).

Summarizing our results, we find that the combination of admission control and careful planning/allocation of cluster resources enables *Rayon* to meet the resource allocation SLAs for all the jobs it accepts (see Figure 10). The results also show that by incorporating work-conserving preemption, *Rayon* is able to maximize cluster utilization (see Figure 11). In contrast, to meet resource allocation SLAs to production jobs, the stock YARN scheduler forces cluster workload to be substantially curtailed, leading to lower utilization. Finally, our results also show that *Rayon* is able to handle the resource demands of a heterogenous mix of jobs (see Figure 14).

In what follows, we describe our experimental setup in Section 4.1 and then present the results of our evaluation.

### 4.1 Experimental setup

Our experimental setup comprises of (1) cluster configuration and the software we deployed (Section 4.1.1) and (2) workloads (Section 4.1.2) used for the evaluation.

#### 4.1.1 Cluster setup

Our large experimental cluster has approximately 256 machines grouped in 7 racks with up to 40 machines/rack. Each machine has 2 8-core Intel Xeon E5-2660 processors with hyper-threading enabled (32 virtual cores), 128GB RAM, 10Gbps network interface card, and 10 2-TB drives configured as a JBOD. The connectivity between any two machines within a rack is 10Gbps while across racks is 6Gbps.

We run Hadoop YARN version 2.1.0 with our modifications for implementing *Rayon*. We use HDFS for storing job input/output with the default 3x replication. On each machine we run an instance of a YARN NM, and an instance

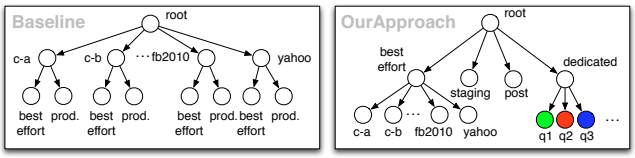


**Figure 9.** Visualization for organization of the queues for our approach and baseline.

of a HDFS data node daemon. The disks on each machine are used to store temporary files (viz., sort spills) as well as HDFS blocks.

With this configuration, to get 100% cluster utilization, we can run a maximum of 25k YARN containers in parallel (or a smaller number of larger containers).

### 4.1.2 Workloads

To evaluate our system we construct synthetic workloads that include (1) jobs with malleable resource needs (viz., MapReduce jobs), (2) jobs with gang-scheduling resource needs (viz., Giraph graph computations), and (3) pipelines with time-varying resource needs (composed of multiple jobs). Each job in the workload is described by an RDL resource expression (see Sections 2.1 and 2.1.1). In what follows, we first describe the methodology we used to construct our workload mix. We then describe how we constructed resource allocation SLAs for production jobs. Finally, we describe how we adapted the workloads to our cluster configuration.

***Distribution-based Map-Reduce Workload*** The SWIM project [7, 8] provides detailed characteristics of MapReduce workloads from (1) five Cloudera customers clusters, (2) two Facebook clusters, and (3) a Yahoo! cluster. The cluster sizes range from 100's of nodes up to 1000's of nodes. For each cluster, the jobs are grouped based on job characteristics such as, I/O patterns, number of map/reduce tasks, and job run time. SWIM provides aggregate information for each group, comprising of an extensive set of statistics. Figure 8 shows the distributions for some of the key parameters.

***Synthetic Giraph Workload*** We use Apache Giraph on YARN to perform page-rank computations on synthetically generated graphs consisting of upto 50 million vertices and approximately 25 billion edges. Such graphs are routinely used for testing purposes at LinkedIn. Recall that Giraph computations require gang-scheduling for their tasks.

***Trace-based Pipelines*** We construct synthetic jobs using the resource profiles collected from a set of production pipelines in Microsoft's clusters. We aggregate the resource usage for a pipeline over its duration, which allows us to construct the resource profile as a "skyline" (see Figure 3). We represent each stage of the "skyline" using a synthetic MapReduce job.

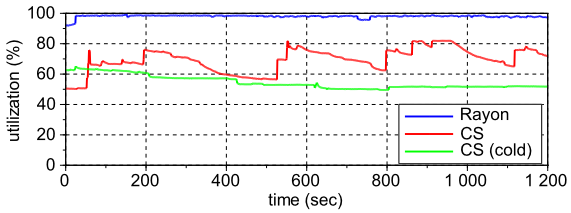


**Figure 11.** Cluster utilization, and preemption over time for: *Rayon*, the CapacityScheduler (CS), and the CapacityScheduler running with reduced load (CS cold).

***SLAs for Production Jobs*** Of the three classes of jobs, the trace-based pipelines were carefully chosen based on conversations with the users who submitted those jobs. They provided information about the periodicity as well as the completion time SLAs for those jobs. However, for the remaining two classes of jobs, we had to estimate both the fraction of the workload mix that corresponded to production jobs as well as their completion time SLAs. Based on our conversations with cluster operators, we estimate that about 5% of the MapReduce jobs require a reservation. We determine job reservations from the actual job duration and parallelism, and account for a 10% slack. Moreover for each job class in SWIM, we determine a plausible deadline, based on job runtime and frequency (see Figure 8).

***Workload Adaptation*** Many of the distribution-based workload traces are from clusters much larger than ours. We adapted the workloads to our cluster by using two techniques. First, we adjusted the rate at which MapReduce jobs are generated. Second, we bounded as well as scaled a job's parallelism (and duration) to allow concurrent execution of jobs.

In the evaluation, we use GridMix (version 3.0) to generate distribution-based MapReduce workloads. We adapt GridMix to submit reservations and created other simple driving infrastructures capable of using the reservation handles to submit multiple Giraph and pipeline jobs.

## 4.2 *Rayon* versus YARN CapacityScheduler

To generate a baseline, we compare *Rayon* versus the stock YARN CapacityScheduler (CS) for the distribution-based MapReduce workload. In the experiment, GridMix generated MapReduce jobs at the rate of 5,400 jobs per hour. For this experiment, we configured the CapacityScheduler queues as follows. For Stock YARN, we created a pair of queues for each of the 8 cluster workloads (i.e., *tenants*) with one queue for best-effort jobs and another for production jobs from that workload (see Figure 9. We tune the guaranteed capacity allocated to each queue to be proportional to the total slot utilization requested by the jobs that will be run in that queue. We follow best practices in configuring maximum capacity for each queue (up to 2x the guaranteed capacity). For *Rayon* we use a rather simple configuration, where 70% of the cluster capacity is dedicated to production

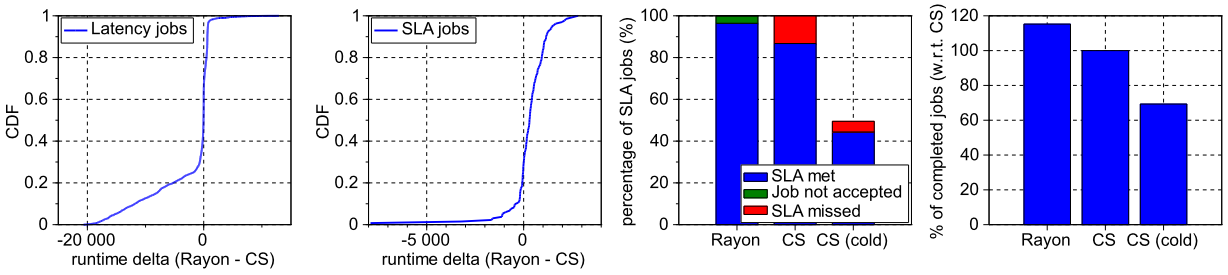


**Figure 10.** End-to-end experiment showing: 1) CDF of job runtime changes for Latency jobs, 2) CDF of job runtime for SLA jobs, 3) bar graph for jobs meeting or missing SLAs, and 4) total number of jobs completed during the experiment.

jobs. The remaining capacity is proportionally allocated to best-effort jobs across the 8 tenants. *Rayon* dynamically divides the 70% portion of the cluster among production jobs, and always redistributes spare resource among production and best-effort jobs.

Figure 10 shows the results of our experiments when running a mix of production and best-effort jobs. We find that *Rayon* meets the resource allocation SLAs to all jobs it admits. Furthermore, by employing admission control, *Rayon* is able to reject jobs for which it cannot meet the allocation SLAs. This feedback, albeit negative, is invaluable to our users. In contrast, the stock YARN capacity scheduler accepts all jobs but fails to meet the SLAs for roughly 15% of them. This means that with the stock YARN capacity scheduler, the cluster has to be under-utilized to increase the odds that SLAs to production jobs are met. In our setting, we found that by reducing the job submission rate by 50% (i.e., 2700 jobs/hour rather than 5400 jobs/hour), production jobs were more appropriately serviced. This corresponds to the "CS(cold)" setting in Figure 10.

Figure 11 shows the cluster CPU utilization between the two systems. The stock YARN CapacityScheduler does not consider preemption. This means that even when there is no compute demand from one queue, idle resources will lie fallow in anticipation of future demand. Consequently, this lowers cluster utilization. On the other hand, since *Rayon* incorporates work-conserving preemption, we are able to maximize cluster utilization. In separate tests, we confirmed that adding preemption to the CapacityScheduler increases cluster utilization, but does not remove SLA violations.

Finally, another metric we can extract from these experiments is the responsiveness of our system in providing the capacity dedicated to a production job. Figure 12 shows the percentage of jobs receiving all their guaranteed capacity within a specifed amount of time. The $x$-axis reports the *sum* of all 1- second windows in which a job was below its SLA with pending resource demand. In the experiments, no job was ever below its resouce allocation SLA for more than 5 seconds in total. Since jobs runtimes are much longer we consider that *Rayon* meets production SLAs.

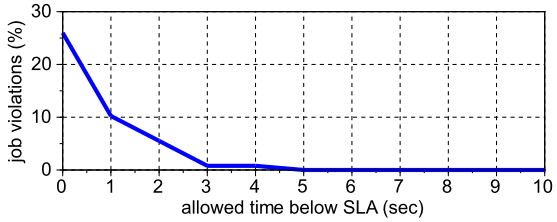


**Figure 12.** Percentage of job violations vs time allowed below nominal SLA.

### 4.3 Handling Mixed Workloads

We now consider a setting in which *Rayon* is used to handle a heterogeneous job mix—(1) SLA jobs with malleable resource demands, (2) SLA jobs with gang scheduling requirements, (3) job pipelines with SLAs, and (4) best-effort jobs. For this experiment we use a mixture of all our workloads.

During this experiment, the Giraph jobs used for (2) did not need to hoard containers, avoiding the associated resource waste, and job pipelines were easily fit in the inventory together with all other production jobs. For Giraph and pipelines we used simple combinations of *any, all* and *order* operators from RDL.

This is shown in Figure 13 where we show the amount of capacity over time for a small subset of the jobs we run [5]. The colors in the figure are used to help distinguish allocation of different jobs/pipelines. In the figure we highlight two particular skylines, which have dramatically varying needs for resources, and clearly come from a periodic invocation of one of the pipelines; periodicity can also be seen for some of the smaller pipelines.

Overall this experiment confirms that *Rayon* is capable of handling a rich mix of jobs, guaranteed SLAs, while maintaining high cluster utilization.

### 4.4 Admission Control: Simulations

In this subsection, we use distribution-based datasets to discuss, explain, validate specific aspects of the Admission Control of *Rayon*.

[5] The normalization is due to the proprietary nature of underlying data

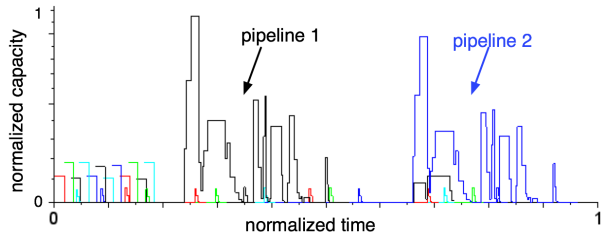


**Figure 13.** Visualization of dynamic queue allocations over time: including pipelines and gangs.

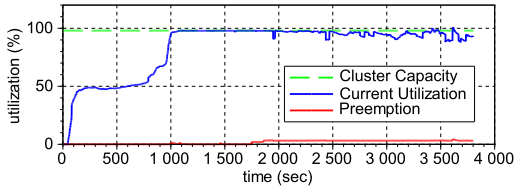


**Figure 14.** Cluster utilization when running a mixture of gang/fluid, besteffort/SLA jobs, and piepelines.

We ran the planning stage in isolation using a 7200 job trace simulating two hours of SWIM workloads. In this environment, a reservation is never removed from the inventory and it is never replanned. In Figure 15, we compare three of the heuristics we designed. The *FifoGang* agent places gangs as early as possible in the inventory. The *Fluid* agent fills available capacity up to the maximum capacity for that job, constrained within a window. The *Skyline* agent places gangs within a window.

For each agent, we consider the effect of cluster size and "slack" in synthetic deadlines on inventory utilization and job acceptance rates. Figure 15 also plots a surface of the configurations simulating the planning stage using the Skyline agent on every job in this workload.

Many of the trends are unsurprising. Generally, less constrained agents accept more jobs and individual agents can accept more work as deadlines slacken. Agents always satisfy more request expressions with a larger inventory, but large clusters can start to dilute utilization if subsequent expansions of capacity accept a smaller fraction of work.

This simulation also illustrates some unintuitive realities of cluster workloads, particularly the effect of very large jobs in the cluster. As cluster size and deadline slack increase, agents may diverge sharply from their trends, and these discontinuities occur inconsistently across agents. For this workload, the Skyline agent often accepts more jobs because it rejects a handful large jobs that are hard to place. The FifoGang agent is particularly sensitive to both cluster size and deadline slack, exhibiting sharp spikes in utilization as it becomes capable of satisfying particular, large request expressions (as there is no corresponding spike in jobs accepted for that cluster size).

The principal lesson we take from these simulations is that greedy, pessimistic planning cannot keep clusters busy

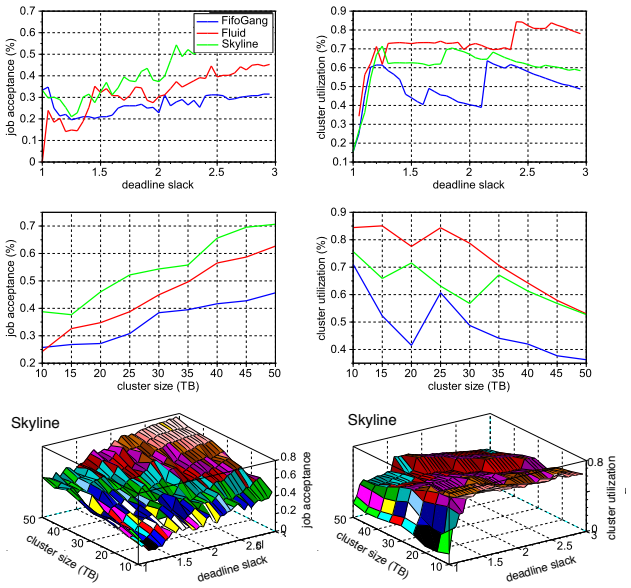


**Figure 15.** Simulating the planning of 7200 jobs/hour.

with these workloads. *Rayon* accepts far more jobs and achieves much higher utilization by running jobs earlier than their reservations, dynamically adapting to cluster changes, and backfilling idle capacity with best-effort jobs.

### 4.5 Impact of Over-reservation

While we do not focus on the resource definition problem in this paper, it is important to understand the impact of wrongly sized reservations. Under-reservation is simple, as the production job will run with guaranteed resources up to a point, and then continue as a best-effort job till completion (thus, subject to uncertainty).

Over-reservation, however, affects job acceptance. In Figure 16, we show what happens to our key metrics if we run the same workload as in Section 4.2, but with jobs over-reserving by 50% (i.e., asking for more resources than they will actually use). The key effects are: 1) job acceptance is reduced (11% of jobs are rejected), 2) SLAs are met for all accepted jobs, 3) cluster throughput for best-effort jobs grows by 20% (as *Rayon* backfills with best-effort jobs), and 4) both SLA and best-effort jobs see improved runtimes. Provided we have enough best-effort jobs waiting for resources, cluster utilization remains close to 100%. This indicates that while overly conservative resource definition can negatively affect job acceptance, and therefore incentives and tools should be created for users to be accurate in describing their resource needs, *Rayon*'s aggressive redistribution of resources allows it to cope with misconfigured reservations and maintain high cluster throughput while still meeting SLAs.

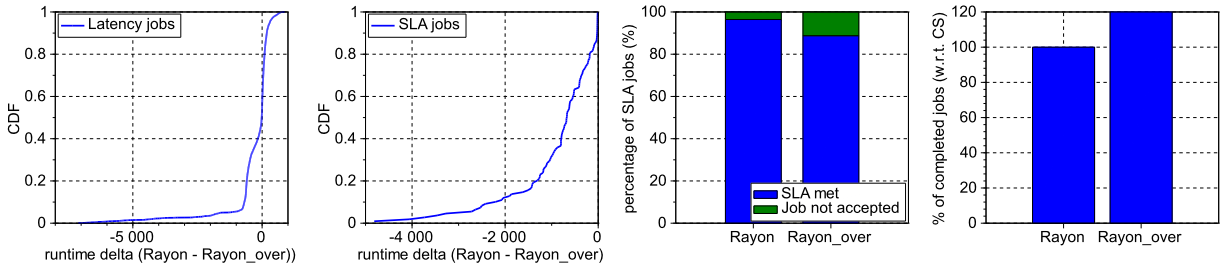


**Figure 16.** Effect of 50% over reservation (i.e., production jobs ask for 50% more resource reservations than they will use).

## 5. Related Work

Resource management for large compute clusters is an area of active research. Delay Scheduling [24], DRF [12], and H-DRF [6] are recent examples of techniques for sharing resources among cluster tenants. Each of these influences or defines a correct allocation given fairness invariants, but none consider how to guarantee its invariants over time. Schedulers supporting compatible workloads such as Quincy [14], Mesos [13], and Omega [21] use job eviction (killing) to improve locality and restore resource invariants in response to changes in the workload, but none of these schedulers have an analogous planning stage for admission control. Quincy and Mesos do not support gang scheduling, while Omega assumes *a priori* coordination of priority among its tenants to ensure that these jobs can acquire and retain their allocation for the necessary duration.

HPC schedulers optimize a similarly shaped space, but against a different set of invariants and for applications with different requirements. Notably, while most applications submitted to YARN or Mesos assume commodity-grade hardware and elastic scaling, HPC applications are usually intolerant of faults and assume gang semantics. As such, their approach to scheduling is very different. One example is SLURM [23], which supports a rich set of algorithms for inferring job priority and mechanisms to evict, suspend, and checkpoint jobs based on those priorities. Unlike *Rayon*, reservations are not logical entitlements, but rather explicit evictions of all work from particular nodes for privileged access (e.g., system maintenance).

Jockey [10] dynamically modifies resource allocations to complete Scope DAGs before their deadlines. Its adjustments are based on an offline simulation modeling the effect of adding resources to the job, but it does not arbitrate among jobs with deadlines, nor consider pipelines, and preemption.

Bazaar [15] provides predictable resource allocation (VMs and network bandwidth) to ensure consistent and timely execution of MapReduce jobs. Fixed, just-in-time reservations are bound to a single job and updated based on the current cluster allocation. Bazaar does not consider preemption. Lucier et al. [17] develop online algorithms for predictable resource allocation assuming work-conserving preemption. Neither approach handles the highly variable allocations of pipelines explicitly; modeling a pipeline deadline by assigning deadlines to its individual computation steps limits the scheduler's flexibility, causing it to be unduly pessimistic and reject pipelines.

Work-conserving preemption for MapReduce computations was previously considered in Amoeba [5] where the focus was solely on mechanisms for work-conserving preemption for Map-Reduce jobs. Natjam [9] also considers job/task preemption policies in the presence of resource contention. The work-conserving preemption mechanisms described in Natjam are similar to those in [5]. However, neither of these systems use admission control, which makes predictable resource allocation hard to guarantee for a particular job. Furthermore, neither system considers jobs with gang-scheduling requirements or job pipelines.

## 6. Concluding Remarks

Ad-hoc clusters dedicated to a single application framework are being progressively replaced by shared multi-framework clusters, creating novel challenges in resource management. Users are coming to expect clusters to support SLAs covering job runtimes, gang allocations, and inter-allocation dependencies. In this paper, we propose a new framework, *Rayon*, to tackle this problem by clearly dividing the responsibility of *defining resource demands* from that of *delivering predictable resource allocation*.

To demonstrate that our *Rayon* framework is practical we implemented it and are open-sourcing it in the context of the Apache YARN system (Hadoop 2.x). We extended the current system by introducing a language for resource reservation, built an admission control component capable of time-aware planning, built and used a work-preserving form of preemption, and extended the underlying scheduling infrastructure by allowing dynamic scheduler reconfiguration.

We evaluated our system on many workloads based on real-world clusters from several companies and demonstrated that by explicitly modeling time, planning conservatively, and leveraging work-preserving preemption, our system can: 1) increase cluster utilization and job throughput, 2) eliminate SLA violations, and 3) increase best-effort job completion times.

Most importantly, we designed our *Rayon* framework to provide an extensible substrate that enables further investigations on planning policies. We are currently exploring the

space of dynamic renegotiation and replanning, value-based planning/pricing and automatic resource definition for jobs and pipelines.

## References


[1] Facebook Corona. http://tinyurl.com/fbcorona.

[2] Apache Giraph Project. http://giraph.apache.org/.

[3] Preemption and restart of mapreduce tasks. https://issues.apache.org/jira/browse/MAPREDUCE-4584.

[4] Hadoop YARN Project. http://tinyurl.com/bnadg9l.

[5] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True Elasticity in Multi-Tenant Clusters through Amoeba. In *ACM Symposium on Cloud Computing*, SoCC'12, October 2012.

[6] Bhattacharya, C. Arka, F. David Culler, G. Eric Friedman, S. Ali, a. S. Scott, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *ACM Symposium on Cloud Computing*, SoCC'13, October 2013.

[7] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 390–399, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4430-4. . URL http://dx.doi.org/10.1109/MASCOTS.2011.12.

[8] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads. *Proc. VLDB Endow.*, 5 (12):1802–1813, Aug. 2012. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=2367502.2367519.

[9] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin. Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters. October 2013.

[10] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 99–112, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3. .

[11] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1:117–129, 1976.

[12] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 24–24, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1972457.1972490.

[13] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1972457.1972488.

[14] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, pages 261–276, 2009.

[15] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. ISBN 978-1-4503-1761-0.

[16] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.

[17] B. Lucier, I. Menache, J. S. Naor, and J. Yaniv. Efficient online scheduling for deadline-sensitive jobs: extended abstract. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, SPAA '13, pages 305–314, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1572-2.

[18] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of mapreduce pipelines. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 681–684. IEEE, 2010.

[19] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Scalable scheduling for sub-second parallel jobs. Technical Report UCB/EECS-2013-29, EECS Department, University of California, Berkeley, Apr 2013.

[20] M. Sarkar, T. Mondal, S. Roy, and N. Mukherjee. Resource requirement prediction using clone detection technique. *Future Gener. Comput. Syst.*, 29(4):936–952, June 2013. ISSN 0167-739X. . URL http://dx.doi.org/10.1016/j.future.2012.09.010.

[21] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013.

[22] V. Vavilapalli and et. al. Apache hadoop yarn: Yet another resource negotiator. In *ACM Symposium on Cloud Computing*, SoCC'13, October 2013.

[23] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[24] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 265–278, 2010.