

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

Carlo Curino<sup>m</sup>, Djellel E. Difallah<sup>u</sup>, Chris Douglas<sup>m</sup>, Subru Krishnan<sup>m</sup>,  
Raghu Ramakrishnan<sup>m</sup>, Sriram Rao<sup>m</sup>  
Cloud Information Services Lab (CISL) — Microsoft Corp.

<sup>m</sup> : {ccurino, cdoug, subru, raghu, sriramra}@microsoft.com, <sup>u</sup>: djelleddine.difallah@unifr.ch

## Abstract

The continuous shift towards data-driven approaches to business, and a growing attention to improving return on investments (ROI) for cluster infrastructures is generating new challenges for big-data frameworks. Systems, originally designed for big batch jobs, now handle an increasingly complex mix of computations. Moreover, they are expected to guarantee stringent SLAs for production jobs and minimize latency for best-effort jobs.

In this paper, we introduce *reservation-based scheduling*, a new approach to this problem. We develop our solution around four key contributions: 1) we propose a *reservation definition language (RDL)* that allows users to declaratively reserve access to cluster resources, 2) we formalize planning of current *and future* cluster resources as a Mixed-Integer Linear Programming (MILP) problem, and propose scalable heuristics, 3) we adaptively distribute resources between production jobs and best-effort jobs, and 4) we integrate all of this in a scalable system named *Rayon*, that builds upon Hadoop / YARN.

We evaluate *Rayon* on a 256-node cluster, against workloads derived from Microsoft, Yahoo!, Facebook, and Cloudera's clusters. To enable practical use of *Rayon*, we hardened our system and open-sourced it as part of Apache Hadoop.

## 1. Introduction

Scale-out computing has enjoyed a surge in interest and adoption following its success at large web companies such as Facebook, Google, LinkedIn, Microsoft, Quantcast, and

Yahoo! [31]. As these architectures and tools become ubiquitous, maximizing cluster utilization and, thus, the return on investment (ROI) is increasingly important.

Characterizing a “typical” workload is nuanced, but the hundreds of thousands of daily jobs run at these sites [35, 39] can be coarsely classified in two groups:

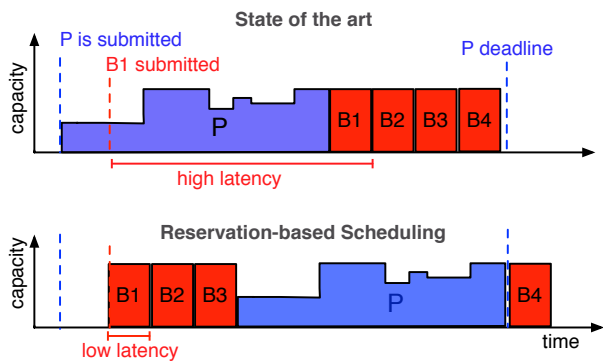
1. *Production jobs*: These are workflows *submitted periodically* by automated systems [26, 37] to process data feeds, refresh models, and publish insights. Production jobs are often *large and long-running*, consuming tens of TBs of data and running for hours. These (DAGs of) jobs are central to the business, and come with strict service level agreements (SLA) (i.e., *completion deadlines*).
2. *Best-effort jobs*: These are ad-hoc, exploratory computations submitted by data scientists and engineers engaged in testing/debugging ideas. They are typically *numerous*, but *smaller* in size. Due to their interactive nature, best-effort jobs do not have explicit SLAs, but are sensitive to *completion latency*.

The mix of jobs from these categories is cluster dependent. Production jobs can be as few as 5% of all jobs. However, in all but dedicated test clusters [10, 11, 13, 39], they consume over 90% of the resources. While numerically few, these jobs are business-critical, and missing SLAs can have substantial financial impact.

Currently deployed big-data systems [2, 23, 39] focus on maximizing cluster throughput, while providing sharing policies based on instantaneous notions of priority, fairness, and capacity. Prioritizing production jobs improves their *chances* to meet SLAs, at the expense of best-effort jobs' latency. Symmetrically, prioritizing best-effort jobs can improve their latency, but it endangers production jobs' SLAs. In either case, unnecessary head-of-line blocking prevents all such time-agnostic mechanisms from simultaneously satisfying the demands of both types of jobs. In particular, *no promises can be made* on jobs' allocations over time.

Interviewing cluster operators, we gather that the above limitations are coped with today by over-provisioning their clusters (detrimental to ROI), or by means of labor-intensive

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.



**Figure 1.** Effectiveness of reservation-based scheduling (A production job  $P$  with time-varying resource needs, and four best-effort jobs  $B_{1..4}$ )

workarounds. These include manually timing job submissions, and ensuring production jobs’ SLAs by dedicating personnel to monitor and kill best-effort jobs if resources become too scarce. 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 [24, 33], where scale and the contractual relationship between users and operators exacerbate these problems.

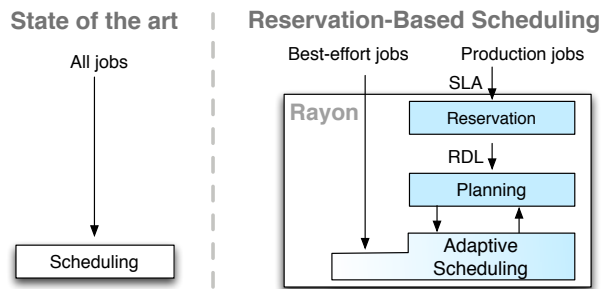
To make matters worse, the days of siloed clusters running a single application framework, such as MapReduce [16], are long gone. Modern big-data clusters typically run a diverse mix of applications [2, 23, 35, 39]. This introduces new scheduling challenges such as supporting gang semantics, and inter-job dependencies.

In this paper we propose **reservation-based scheduling**, a novel approach that delivers time-predictable resource allocations to: 1) meet production job SLAs, 2) minimize best-effort job latency, and 3) achieve high-cluster utilization.

**Contributions** Our effort builds upon ideas from extensive prior work on big-data frameworks, HPC infrastructures, and scheduling theory, [2, 3, 18, 19, 23, 27, 29, 35, 36, 38–41], but provides a unique combination of features including support for a rich constraint language, scalable planning algorithms, and adaptive scheduling mechanisms. We integrated all of this in a complete architecture and robust implementation that is released as part of Apache Hadoop. To the best of our knowledge, our system, *Rayon*, is the first big-data framework to support completion SLAs, low latency, and high-cluster utilization for diverse workloads at scale.

Our effort is organized around four key contributions (visualized in Figure 2):

1. *Reservation*: this is the process of determining a job’s resource needs and temporal requirements, and *translating the job’s completion SLA into a service level objective (SLO) over predictable resource allocations*. This is done ahead of job’s execution and it is akin to a *reservation* of resources, aimed at ensuring a predictable and timely execution. To this end, we propose a Reservation Definition Language (RDL), that can express in a declaratively fash-



**Figure 2.** Overview of our approach.

ion a rich class of constraints, including deadlines, malleable and gang parallelism requirements, and inter-job dependencies—see Section 2.

2. *Planning*: RDL provides a uniform and abstract representation of all the jobs’ needs. Such reservation requests are received by the system ahead of a job’s submission. We leverage this information to perform *online admission control*, accepting all jobs that can fit in the cluster agenda, aka the *Plan*, and rejecting the ones we cannot satisfy. In Section 3, we formalize *Planning* as a Mixed-Integer Linear Programming (MILP) problem, and propose robust and scalable greedy algorithms.
3. *Adaptive Scheduling*: this is the process of dynamically assigning cluster resources to: 1) production jobs, based on their allocation in the plan, and 2) best-effort jobs submitted on the fly to minimize their latency. In this phase, we dynamically adapt to the evolving conditions of a highly-utilized, large cluster, compensating for faults, mispredictions, and other system imperfections—see Section 4.
4. *Rayon*: Our final contribution is to integrate the above ideas in a complete architecture. We instantiate our design in a YARN-based system [39]. Over the past year we have hardened our system and open-sourced<sup>1</sup> it as part of Apache Hadoop—Section 4.2. We validate *Rayon* on a 256-node cluster, running jobs derived from Microsoft clusters, and workloads derived from clusters of Cloudera customers, Facebook, and Yahoo!—Section 5.

By introducing the notion of reservation, we arm *Rayon* with substantially more information to handle jobs with SLAs. We illustrate this pictorially in Figure 1, where we delay  $P$ , meet its SLA, and improve  $B_1, B_3, B_3$ ’s latencies. Our experimental evaluation confirms that this extra information, combined with effective planning, and adaptive scheduling delivers an advantage over existing approaches.

The rest of this paper discusses these contributions. This presentation is necessarily terse, but more details and experimental evidence can be found in the companion technical report [14]. We track our open-sourcing effort at <https://issues.apache.org/jira/browse/YARN-1051>.

<sup>1</sup> Our contributions are being code reviewed, and are likely to be committed to Hadoop 3.0 by camera-ready time.

## 2. Reservation

As mentioned above, the types of computations that are run on modern big-data clusters have diversified from MapReduce jobs to interactive analytics, stream and graph processing, iterative machine learning, MPI-style computations [10, 11, 13], and complex workflows (DAGs of jobs) leveraging multiple frameworks [8, 37]. Moreover, the consolidation of clusters means that production jobs with strict deadlines will be run together with latency critical best-effort jobs. In this section, we focus on designing a reservation definition language (RDL) capable of expressing the above.

We distill the following key requirements:

- R1** malleability for batch jobs (e.g., MapReduce)
- R2** strict parallelism and continuity for gang jobs (e.g., MPI)
- R3** explicit temporal requirements (i.e., SLAs).
- R4** precedence constraints (dependencies) among jobs that comprise a pipeline (e.g., Hive, Oozie, Azkaban)
- R5** expose all of the placement flexibility.

This set of requirements captures most of the practical scenarios we encountered. We formalize RDL next.

### 2.1 Reservation Definition Language (RDL)

An RDL *expression* can be:

1. An *atomic expression* of the form  $\text{atom}(b, g, h, l, w)$ , where:  $b$  is a multi-dimensional bundle of resources<sup>2</sup> (e.g., <2GB RAM, 1 core>) representing the “unit” of allocation,  $h$  is the maximum number of bundles the job can leverage in parallel,  $g$  is the minimum number of parallel bundles required by the job; a valid allocation of capacity at a time quanta is either 0 bundles or a number of bundles in the range  $[g, h]$ .  $l$  is the minimum lease duration of each allocation; each allocation must persist for at least  $l$  time steps, and  $w$  is the threshold of work necessary to complete the reservation (expressed as bundle hours); the expression is satisfied iff the sum of all its allocations is equal to  $w$ . In Figure 3a, we show an example of atomic expression.
2. A *choice expression* of the form  $\text{any}(e_1, \dots, e_n)$ . It is satisfied if *any* one of the expressions  $e_i$  is satisfied.
3. A *union expression* of the form  $\text{all}(e_1, \dots, e_n)$ . It is satisfied if *all* the expressions  $e_i$  are satisfied.
4. A *dependency expression* of the form  $\text{order}(e_1, \dots, e_n)$ . It is satisfied if for all  $i$  the expression  $e_i$  is satisfied with allocations that strictly precede all allocations of  $e_{i+1}$ .
5. A *window expression* of the form  $\text{window}(e, s, f)$ , where  $e$  is an expression and  $[s, f]$  is a time interval. This bounds the time range for valid allocations of  $e$ .

It is easy to see that RDL allows users to express completely *malleable* jobs such as MapReduce (by setting  $g = 1$  and  $l = 1$ ) and very *rigid* jobs such as MPI computations requiring uninterrupted and concurrent execution of all their tasks (by setting  $g = h$  and  $l = w/h$ )—requirements **R1**,

<sup>2</sup>This match closely YARN containers [39], and multi-resource demands of [21].

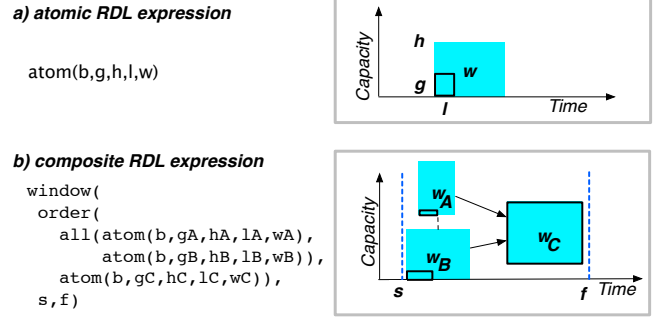


Figure 3. Two RDL expressions

**R2**. The `window` operator allows to constrain the interval of validity of any sub-expression, its natural application is to express completion deadlines—requirement **R3**. Users can represent complex pipelines and DAGs of jobs in RDL using the `order`, `all`—requirement **R4**. The `any` operator allows to express alternative options to satisfy a single reservation.

Note that RDL expressions typically admit multiple solutions for a given plan—requirement **R5**. Choosing between equivalently valid allocations is a prerogative of the *Planning* phase (discussed in Section 3), which leverages this flexibility to optimize system-wide properties such as efficiency of resource utilization, fault resilience, etc.

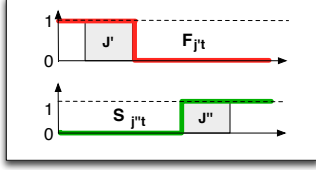
Finally, while best-effort jobs do not participate in the reservation process they could be formally represented by a void atomic expression  $\text{atom}(b, 1, h, 1, 0)$ , which is trivially satisfied; Such void expression provides no “guaranteed” access to resources, as  $w = 0$ , but only best-effort access to idle resources.

We illustrate the use of RDL with the hypothetical production workflow of Figure 3 (b). This workflow is composed of three jobs, two malleable batch jobs  $\{A, B\}$  (e.g., MapReduce), and one  $\{C\}$  with a gang requirement (e.g., Giraph [1]).  $C$  execution depends on the  $A$  and  $B$ .

**RDL completeness and limitations** We do not make completeness claims about the RDL language, but we find it sufficient to naturally capture all common practical scenarios we encountered. We broadly validated this claim by socializing our language design with the Apache Hadoop community (a large group of users of big-data systems), as part of *Rayon*’s open-sourcing. In general, we found strong support for using RDL as the reservation language for Hadoop. An existing limitation of RDL is the lack of support for relative time constraints, and periodic expressions. We are considering to extend RDL based on user feedback.

**Deriving RDL from user jobs** We conclude this section by discussing how users (or tools on their behalf) can derive RDL expressions for their jobs. Most production jobs naturally fall into one of the following well behaved categories:

*High-level frameworks*: Despite the diversity of workflows a significant fraction of production jobs are generated by a handful of frameworks such as Hive, Pig, Giraph [10, 11, 13]. This gives us an opportunity to build profil-



**Figure 4.** order: start and finish support functions.

ers and optimizers capable of automatically producing precise resource demands for queries/jobs by leveraging the application semantics [17–19, 27, 32, 40]. In particular, we extended [17] to generate RDL expressions from Hive and MapReduce jobs, and we are currently collaborating with the authors of [32] to port their Giraph-centric solution to *Rayon*. We plan to report on this in follow up work.

*Periodic jobs:* Production jobs are often canned computations, submitted periodically [13, 19]. This makes them amenable to history-based resource prediction [34]. We informally validate this by running seasonality detection and building predictors for workflows from internal Microsoft clusters. We report on this in Section 5.2.

Socializing RDL with the OSS community we gathered interest in using it as a dynamic provisioning tool to provide time-evolving allocations to users. All of above amounts to an informal but promising validation of RDL practicality. We discuss this in details in [14].

**Compiling RDL: normalization and optimization** RDL expressions are processed by a compiler/optimization layer we purpose built. Our compiler automatically verifies simple satisfiability, and optimizes the input expressions, by applying a series of semantics-preserving transformations such as: 1) redundant operators removal 2) unification of compatible atomic expressions and 3) operators re-ordering (push-down of window operators). This process gives users complete freedom in expressing their requirements with RDL, but simplifies and accelerates our placement algorithms. The compiler produces an internal AST representation, that is used to generate the MILP formulation we discuss next, as well as, structured and normalized input formats to the heuristics of Section 3.2.

### 3. Planning

The RDL expressions define the temporal resource needs of jobs. Planning the cluster’s agenda involves constructing a temporal assignment of cluster resources to jobs such that each job’s RDL expression is satisfied. Given the expressivity of RDL this planning problem is provably NP-complete<sup>3</sup>.

We formalize such planning as a combinatorial optimization problem (Section 3.1). The resulting formulation is a Mixed-Integer Linear Program (MILP) that covers all features of RDL (and can be generated by our compiler).

<sup>3</sup> See [14].

#### 3.1 Formalizing planning as an optimization problem

For the sake of presentation we omit the nesting properties of RDL and introduce the model progressively.

**Basic formulation** We use variables  $x_{jt}$  to represent the resources allocated to job  $j$  during time step  $t$ . Using these variables, we can formulate linear inequalities to assert packing and covering constraints on feasible allocations, and define a objective function as follows:

$$\text{minimize } \sum_{j,t} c_{jt} * x_{jt}$$

subject to:

$$\forall t : \sum_j x_{jt} \leq Cap_t \quad (1)$$

$$\forall j \forall t : x_{jt} \leq h_j \quad (2)$$

$$\forall j : \sum_{s_j \leq t < f_j} x_{jt} = w_j \quad (3)$$

$$\forall j, t : x_{jt} \in \mathbb{R}_+ \quad (4)$$

where, the allocation of  $x_{jt}$  are positive real variables (4) subject to the following constraints:

- (1) *capacity constraint:* at every time  $t$  the sum of allocations must be within the physical cluster capacity<sup>4</sup>,
- (2) *parallelism constraint:* the resources allocated to a job are bounded by the job’s maximum parallelism  $h_j$ , and
- (3) *demand constraint:* the resources allocated between a job’s start and completion time satisfy its demand  $w_j$ .

This formulation covers `atom` expression in RDL (except gang semantics discussed next), and window operators. In this basic formulation, the model is infeasible if all jobs cannot be fit. We address this issue in Section 3.1.1. Subject to these constraints we minimize the cost of the overall allocation, expressed as a weighted sum over all allocations.  $c_{jt}$  captures the cost of assigning capacity to job  $j$  at time  $t$ . Therefore, controlling the assignments of  $c_{jt}$  allows us to: 1) prioritize allocations of jobs, and 2) make certain time periods more expensive. To cover the full semantics of RDL’s `atom` operator we extend our basic formulation as follows.

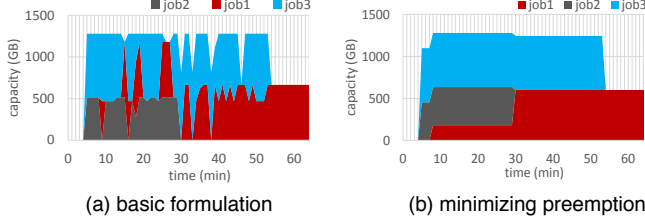
**Supporting gang semantics**  $g$  We support gang semantics by changing (4) for the set of jobs with gang semantics as:

$$\forall j \notin Gang, \forall t : x_{jt} \in \mathbb{R}_+ \quad (5)$$

$$\forall j \in Gang, \forall t : x_{jt} \in \{0, g_j, 2g_j, 3g_j, \dots, h_j\} \quad (6)$$

where *Gang* is the set of jobs with gang semantics, i.e.,  $j \in Gang \iff g_j > 1$ . Assuming  $h_j$  is an exact multiple of  $g_j$  (6) above forces allocations to respect the gang requirements. Supporting gangs, as well as order, any, forces our problem in Mixed-Integer Linear Programming (MILP) territory. We quantify the computational cost of dealing with integrality in Section 5.3.

<sup>4</sup> We simplify our multi-resource formulation for the sake of presentation.



**Figure 5.** Preemption of our allocations.

**Supporting minimum lease duration  $l$**  The minimum lease duration requirement expresses the need for allocations that last at least  $l$  time steps. We start by assuming full rigidity, i.e.,  $g_j = h_j$  and  $l_j = \frac{w_j}{h_j}$ . Under this assumption every valid allocation must have *exactly one transition up* from 0 to  $g_j$  and after  $l$  time steps *exactly one transition down* from  $g_j$  to 0. We can force allocations to assume this shape, by introducing a set of support variables  $y_{jt}$  bound to assume the absolute value of the discrete derivative of  $x_{jt}$ , i.e.,  $y_{jt} = |x_{jt} - x_{j(t-1)}|$ , and then constrain their sum to allow only one transition up and one down:

$$\forall j \in \text{Gang} \sum_t y_{jt} \leq 2 * g_j \quad (7)$$

This combined with (3) and (6) forces each allocation to last precisely  $l$  time steps. Note that the absolute values can be linearized, by expressing them as:

$$\forall j \forall t : y_{jt} \geq x_{jt} - x_{j(t-1)} \quad (8)$$

$$\forall j \forall t : y_{jt} \geq x_{j(t-1)} - x_{jt} \quad (9)$$

This works because we are minimizing  $y_{jt}$ , and only one of the constraints (8) or (9) will be active for any given assignment of  $x_{jt}$  and  $x_{j(t-1)}$ . The full rigidity assumption we made on the atom expressions ( $g_j = h_j$  and  $l_j = \frac{w_j}{h_j}$ ) can be lifted by means of RDL rewriting. Our compiler automatically rewrites each atom expression that requires gangs but is not fully rigid in a more complex  $\text{all}(\text{order}(\dots, a_i, \dots)^*)$  expression where each of the  $a_i$  atoms is fully rigid.

Note that even ignoring the integrality of (6), the constraints (8) and (9) have negative coefficients, therefore we cannot use the fast solver techniques applicable to classical packing and covering constraints [42].

**Supporting any and all** The RDL any operator allows us to express that two (or more) sub-expressions are alternatives, where the system is free to pick one. We capture each sub-expression as a separate job in the formulation, and then constrain their placement. To support this we employ a classical trick in optimization, which is to introduce a slack/overflow variable  $x_{jo}$  for each job in constraint (2). We then introduce an integral variable  $o_j$  that is set to 1 if  $x_{jo}$  is greater than zero, as follows:

$$\forall j : x_{jo} + \sum_{b_j \leq t < e_j} x_{jt} = w_j \quad (10)$$

$$\forall j : o_j > \frac{x_{jo}}{w_j} \quad (11)$$

Intuitively, if  $o_j$  is equal to 1 the corresponding job  $j$  (one of the any alternatives) is not placed. We then impose that for  $k$  jobs tied by a single any expression all but one of the  $o_j$  to be 1. This forces the solver to pick exactly one of the any alternatives. The same can be done for all, simply forcing the sum to zero (i.e., all the jobs must be placed)—the need for this will be apparent later.

**Supporting order** RDL allows us to express temporal dependencies among allocations:  $\text{order}(e_1, \dots, e_n)$ . The intuition behind supporting this in our MILP formulation is to define, a set of support variables  $s_{jt}$  and  $f_{jt}$ , that represent the “start” and “finish” of each sub-expression  $j'$ ,  $j''$ , and then impose that allocations of  $j''$  start after allocations of  $j'$  finish. This is achieved by constraining the relative values of  $s_{j''t}$  and  $f_{j't}$ .

This newly introduced variables must be integral, with  $s_{jt}$  transitioning from 0 to 1 at the first non-zero allocation of  $x_{jt}$ , and  $f_{jt}$  transitioning from 1 to 0 at the last non-zero allocation for  $x_{jt}$ . This is shown pictorially in Figure 4, and defined formally as:

$$\forall j \forall t : s_{jt} \geq s_{j(t-1)} \quad (12)$$

$$\forall j \forall t : s_{jt} \geq \frac{x_{jt}}{h_j} \quad (13)$$

$$\forall j \forall t : f_{jt} \geq f_{j(t+1)} \quad (14)$$

$$\forall j \forall t : f_{jt} \geq \frac{x_{jt}}{h_j} \quad (15)$$

$$\forall (j', j'') \in D \forall t : s_{j''t} \leq 1 - f_{j't} \quad (16)$$

$$\forall j \forall t : s_{jt}, f_{jt} \in \{0, 1\} \quad (17)$$

where  $D$  is the set of dependencies. Note that to express ordering among jobs with different max parallelism  $h_j$ , we normalize constraint (13) and (15), and impose integrality (17) for  $s_{jt}$  and  $f_{jt}$ . Finally, constraint (16) imposes that the first non-zero allocation of  $j''$  must happen after the last non-zero allocation for  $j'$ . Supporting order substantially increases the number of integral variables and constraints, and we will see is a key limiting factor for the practical use of our MILP formulation.

### 3.1.1 Improving our formulation

What we discussed so far covers the semantics of RDL, now we turn to improving the quality of our solutions by capturing important practical considerations.

**Minimizing preemption** We focus on improving the quality of the allocations, by extending our objective function. When the plan allocations change drastically from one time step to the next, the underlying system must quickly redistribute the physical resources among jobs. This requires

the use of preemption [12, 39], and incurs overhead. We minimize abrupt vertical transitions by introducing a term  $\sum_{jt} y_{jt}$  in our objective function, i.e., minimizing the absolute value of derivatives. Figure 5 shows the improvement delivered by this addition by running a commercial solver on a 3-job instance of this MILP formulation.

**Avoiding infeasible models** The formulation we described so far requires that *every* job is assigned by its deadline. This can be troubling, as an MILP solver would return an infeasible model error if it cannot place all of the jobs. Pragmatically we expect this to happen frequently, and prefer a more graceful degradation, where as many jobs as possible are placed, and only few rejected. We leverage the notion of overflow variables we introduced to support the `any` and `all` semantics, but instead of imposing a hard constraint on the sum of the  $o_j$  we modify the objective function. We make it very expensive to use the overflow variables by adding the following term to our objective function:  $\sum_j \alpha_j * o_j$ , with  $\alpha_j$  being a weighting factor that describe how bad it is to reject job  $j$ .

### 3.1.2 Discussion

*Rayon* makes use of planning in two ways: *online*, to commit to accept/reject jobs on arrival (i.e., as soon as their reservation requests are received), and *offline*, to reorganize sets of (already accepted) jobs, optimizing their allocations, possibly in response to changing cluster conditions.

The MILP formulation discussed so far is a useful formal tool, and by leveraging powerful commercial solvers (such as Gurobi) we use it to study the solution space. However, it is not practical for online scenarios, and cannot scale to large problem sizes. This is due to the corresponding explosion of the number of variables and constraints. The practical limit, even for offline uses, is hundreds of `atom` expressions, or tens of complex `order`, `any`, `all` expressions. Furthermore, there are prior impossibility results on designing optimal algorithms for commitment on arrival scheduling problems [29]. For these reasons, we focus on greedy heuristics next.

### 3.2 Greedy Heuristics

The algorithms we present are greedy in two dimensions. First, they place one job at a time, and never reconsider placement decisions for previously placed jobs. Second, as they traverse an RDL expression, sub-expressions are placed with no backtracking. This has some impact on the number of jobs these policies can accept, but placement is scalable and fast. We study this trade-off experimentally in Section 5.3.

**Procrastinating heuristic (GREE)** Our first placement heuristic places a job as close to its deadline as it can, as shown in Figure 6. This is done by traversing the AST representation of the RDL expression of each input job in a one-pass, right-deep, depth-first fashion. Each sub-expression  $e$  is placed (compatibly with its constraints and space avail-

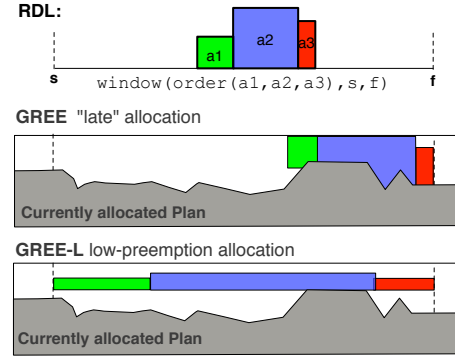


Figure 6. Visualizing GREE and GREE-L allocations.

---

#### Algorithm 1: GREE-L (low preemption greedy)

---

```

Input: Plan  $p$ , RDL  $e$ , TimeInterval  $ti$ 
Result: An assignment of  $e$  in  $p$ 
switch  $e.type$  do
  case (window)
    GREE-L ( $p, e, ti \cap e.window$ );
  case (all, any, order)
    foreach ( $RDL\ subExpr : reverseChild(e)$ ) do
      TimeInterval  $l_{ti} = guessInterval(subExpr)$ ;
      GREE-L ( $p, subExpr, l_{ti}$ );
    return  $p.curAlloc$ ;
  case (atom)
    foreach  $Time\ t:ti\ do$ 
       $p.assign(e, ti, e.w/ti.length)$ ;
return  $p.curAlloc$ ;

```

---

ability) as late as possible. To place an atomic allocation we scan the plan right to left and track the maximum available height (up to  $h$ ) and the most constrained point in time  $t_{lim}$ . When we reach  $l$  consecutive instants in time where the height of the allocation exceeds  $g$ , we allocate this portion of work. If work remains to be allocated, we restart the search at  $t_{lim} - 1$ . Intuitively, this finds the tallest, rightmost allocation for this plan that is compatible with the expression constraints. We enforce order constraints by updating the time range in which we place preceding atomic expressions. For any expressions we behave greedily and accept the first alternative that fits (and never backtrack).

Allocating “late” may appear counter-intuitive (*why would one allocate late when the job might fit in earlier parts of the plan?*). In practice, this policy improves the chances of jobs that show up late but have an early deadline to be placed, and works surprisingly well in coordination with the underlying adaptive scheduling mechanisms that we discuss in Section 4. In fact, the allocations produced by the planner prescribe the *guaranteed* access to resources for a job, while the underlying adaptive scheduler allows jobs to exceed their guaranteed allocations if there are idle resources (redistributing resources based on weighted fairness).

When running many best-effort and SLA jobs in a cluster the effect of this lazy planning and eager scheduling is

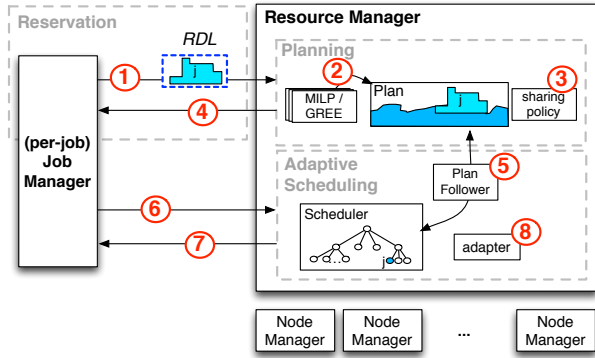


Figure 7. Overview of the *Rayon* system.

to give good latency to best-effort jobs, while still meeting all SLAs—we evaluate these claims experimentally in Section 5.4.

**Lower preemption heuristics (GREE-L)** As we show in Figure 5(b), smoother allocations are preferable as they incur less preemption. GREE is rather bad from this point of view, as it tends to produce “tall and skinny” allocations. We thus propose GREE-L a variant of GREE that trades jobs’ acceptance in exchange for reduced preemption. The pseudo-code for the GREE-L algorithm is shown in Algorithm 1.

The *guessIntervals* function divides the valid time for each expression into  $K$  time intervals (one for each child expression). Like GREE, the traversal proceeds right to left (*reverseChild*), but assignment is done for the atom expressions such that the allocation is as “flat” as possible throughout the heuristically selected interval. We show this in Figure 6. Each time an expression is placed, the sub-intervals are recomputed (redistributing the left-over space).

Rejection is not shown in Algorithm 1 due to space constraints, but it matches the operator semantics described in section 2.1. GREE-L rejects more jobs than GREE because it does not backtrack when allocating flatter reservations in the plan; its subsequent placement of early stages may be infeasible, due to sparser use of the area closer to the deadline.

Note that both GREE and GREE-L might reject jobs that the MILP formulation accepts, as they do not consider moving previously accepted jobs or stages. On the other hand, they are very fast and scalable and accept a competitive fraction of production jobs, as illustrated in Section 5.3.

## 4. Adaptive Scheduling / Rayon architecture

In this section, we describe an architecture for a fully functional big-data system that leverages RDL and the Planning algorithms of Section 3. We also present a form of Adaptive Scheduling designed to cope with practical concerns that emerge from real-world scenarios, including: scaling to thousands of machines and hundreds of thousands of daily jobs, supporting user quotas, and handling of failures, mis-predictions, and systematic biases. We describe the architec-

ture in general terms (Section 4.1), but the reader familiar with any modern big-data system (YARN, Mesos, Omega, or Corona) should notice obvious similarities to those architectures. We make them explicit in Section 4.2 where we cast our design as an extension of YARN [39].

### 4.1 Design

With reference to Figure 7, the architecture we propose contains the following components: 1) a central *resource manager* arbitrating the allocation of physical resources to jobs, 2) *node managers* running on each worker node, reporting to the resource manager liveness information and enforcing access to local resources, and 3) per-job *job managers* negotiating with the resource manager to access resources on the worker nodes, and orchestrate the job’s execution flow<sup>5</sup>.

Following Figure 7, we present the next level of detail by discussing the steps involved in running production (and best-effort) jobs:

- Step 1* The job manager estimates the demand generated by a production job (see Section 2.1) and encodes its constraints as an RDL expression, submitted to the resource manager at *reservation time*.
- Step 2* The *Planning* component of the resource manager maintains a *Plan* of the commitments made on cluster resources by tracking all admitted reservations. This component leverages a set of pluggable placement policies (i.e., the MILP, GREE and GREE-L algorithms of Section 3), to determine whether and how the RDL expression can fit in the current *Plan*.
- Step 3* The resulting allocation is validated both against physical resource constraints and sharing policies. Sharing policies enforce time-extended notions of user quotas.
- Step 4* The user receives immediate feedback on whether the RDL request is accepted. Accepted RDL expressions are tracked by the *Plan*, and define a *contract* between users and the system.
- Step 5* The *Scheduler* is in charge of dispatching resources to jobs, tracking detailed locality preferences, and enforcing instantaneous invariants such as fairness, capacity and priorities. A component called *PlanFollower*, monitors cluster conditions and translates the absolute promises we made in the *Plan* to the relative terms of the underlying *Scheduler* (e.g., increasing a jobs’ priority).
- Step 6* When a job starts its execution, the runtime component of the job manager requests resources based on instantaneous needs from the job. Production jobs specify their reservation ID, and the *Scheduler* guarantees that they will receive at least the resources reserved for that contract. Idle resources are redistributed according to fairness/capacity semantics [21, 39] among both production and best-effort jobs.

<sup>5</sup>Note that the architecture, like any modern big-data system, allows for arbitrary application frameworks (e.g., MapReduce, Giraph, Spark, REEF).

*Step 7* The job manager receives access to resources and proceeds to spawn the tasks of the job as processes running on the worker nodes controlled by the *node managers*.

*Step 8* The *adapter* and the PlanFollower of *Step 5* are a key component of our adaptive scheduling approach. The adapter dynamically rearranges the Plan in response to changes in cluster capacity (node failures or additions). This runs the Planning algorithms in an offline mode, where all accepted jobs are placed in the Plan anew.

*Repeat* During job execution job managers might detect that the application-level progress is happening faster/slower than foreseen at reservation time, and wish to change its reservation. The API we expose to the job manager allows for dynamic *renegotiation* of reservations

In the preceding, every RDL expression is associated with a single job. More generally, each RDL reservation can support a *session* accepting an arbitrary number of jobs. Next, we discuss how this architecture is implemented in the YARN codebase and provide details on the PlanFollower, Adapter, and Sharing policies.

## 4.2 YARN-based implementation

The structure of our architecture is largely compatible with each of the recent big-data systems [2, 23, 35, 39]. We chose YARN [39] as the starting point for our implementation due to its popularity, availability as an open-source project, and our familiarity with the platform<sup>6</sup>.

**Protocol and architectural changes** In order to support *Step 1* and *Step 4* we modify YARN’s application submission protocol, by introducing four new APIs for reservation:

YARN Protocol Extension	
API call	return value
createRes(ResDefinition rdl)	ResID
updateRes(ResID curRes, ResDefinition rdl)	boolean
deleteRes(ResID curRes)	boolean
listRes(UserID userResID)	List<ResID>

This allow users and tools to reserve resources ahead of execution, and to dynamically update this reservation (i.e., the renegotiation steps discussed above). In order to support this new API, we extended YARN’s ResourceManager substantially, by introducing the Planning layer of Figure 7 a new component to the YARN’s architecture. This includes a scalable representation of a Plan (capturing allocations in a compact run-length encoded form), and fast implementations of GREE and GREE-L for online acceptance of RDL expressions (*Step 2*).

**Sharing Policies** The notion of sharing policy is derived from conversations with professional cluster administrators that expressed the need to govern and bound the freedom given to users by RDL. Without a sharing policy (such as user quotas), *Rayon* would allow a user to ask for arbitrary allocations of resources, subject only to physical constraints. These policies are pluggable, and we provide a first imple-

mentation that extends the classical notion of capacity to express constraints over the both integral and instantaneous resources.

**PlanFollower** Both Planning and Scheduling track resource availability and job demand, but they do so in substantially different ways. Planning provides an explicit notion of time, manages demands at the job level, and resources as an undiversified continuum. In contrast, Scheduling focuses only on the current slice of time, but handles demands at a task level and resources at a node level. This two-level view is a fundamental design point to limit the complexity of each component. The PlanFollower (*Step 5*) is the key to translate between these two worlds. Mechanically the PlanFollower runs on a timer, reads the Plan current state and updates the Scheduler configuration to affect the resources that will be given to each job during *Step 6* and *Step 7*. In YARN this required us to modify the CapacityScheduler and FairScheduler to allow for dynamic creation/destruction/resizing of queues—YARN’s mechanism to partition resources among user groups [39].

**Adapter** Planning the use of future resources is at odds with the reality of fast-evolving conditions in large clusters (frequent node failures), errors in the user supplied reservation requests, and imperfections in the underlying infrastructure. In our implementation, we cope with this by implementing the Adapter component of *Step 8*. This consists of a software module actively monitoring the cluster conditions, comparing them with the expectations we have on future resources, and triggering re-planning actions as required. The adapter is also in charge to cope with systematic biases, such as scheduling delays for large tasks (a known limitation of the CapacityScheduler [39]).

## 5. Experimental Evaluation

In this experimental evaluation we validate our hunches on RDL expressivity and usability (Section 5.2), analyze the quality and complexity of Planning, comparing our MILP formulation and greedy algorithms (Section 5.3), and test our end-to-end design on a large and busy 256 machines cluster, comparing it against stock YARN on previously published workloads [10, 11] and production jobs from Microsoft clusters (Section 5.4).

The key insight we obtain can be summarized as follows:

1. RDL naturally is a practical and reasonably easy to use language;
2. for large clusters, our greedy algorithm GREE-L matches the quality of solutions of MILP (i.e., high job acceptance rates, and low preemption). GREE-L is up to 5 orders of magnitude faster than MILP while placing complex workloads.
3. Adaptive scheduling allows us to achieve cluster utilizations approaching 100%.

<sup>6</sup> We leverage our experience with the codebase and the prior work on preemption [39] that we integrated and contributed to Apache.



4. *Rayon* reliably meets the SLAs of 100% of accepted jobs, improves throughput by 15% and delivers better latency to 40% of best-effort jobs.

These results are due to two main factors: 1) by introducing the notion of *reservation-based scheduling*, we arm *Rayon* with inherently more information about the jobs it runs, and 2) our algorithms and system implementation leverage this advantage effectively.

Therefore we conclude that: *Introducing an explicit representation of time, reservation-based scheduling significantly improves predictability in running a mix of production and best-effort jobs, enabling cluster operators to make promises on jobs' allocation over time.*

We differ further tests and more detailed analysis of the workloads to our technical report [14].

## 5.1 Experimental setup

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

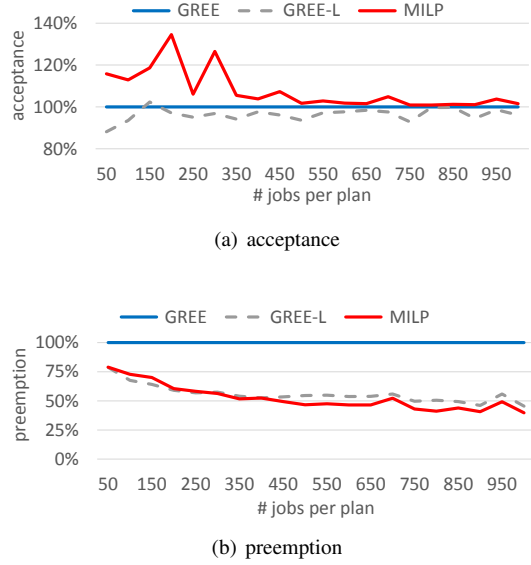
**Cluster setup** Our large experimental cluster has approximately 256 machines grouped in 7 racks with up to 40 machines/rack. Each machine has 2 X 8-core Intel Xeon E5-2660 processors with hyper-threading enabled (32 virtual cores), 128GB RAM, 10Gbps network interface card, and 10 X 3-TB data drives 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.x with our modifications for implementing *Rayon*. We use HDFS for storing job input/output with the default 3x replication. We use a Gurobi 5.6 parallel solver running on a 128GB RAM, 32 cores server, whenever a MILP solver is needed.

**Workloads** To evaluate our system we construct synthetic workloads that include 1) jobs with malleable resource needs (e.g., MapReduce jobs), 2) jobs with gang-scheduling resource needs (e.g., Giraph graph computations), and 3) workflows with time-varying resource needs (e.g., Oozie/Hive). These are respectively derived from:

**Workload A: distribution-based Map-Reduce Workload** The SWIM project [10, 11] provides detailed characteristics of workloads from five Cloudera customers clusters, two Facebook clusters, and a Yahoo! cluster. The cluster sizes range from 100's of nodes up to 1000's of nodes. We devised a synthetic generator based on Gridmix 3.0, producing jobs that respect the original distributions of: submission time, job counts, sizes, I/O patterns, and task runtimes.

**Workload B: Giraph jobs with gang semantics** We use Apache Giraph to perform page-rank computations on synthetically generated graphs consisting of up to 50 million vertices and approximately 25 billion edges. We base this on graphs that are routinely used for testing purposes at LinkedIn. Recall that Giraph computations require gang-scheduling for their tasks.



**Figure 8.** Comparing MILP, GREE, and GREE-L, for increasing number of jobs and increasing cluster size.

**Workload C: Traces of production workflows** We construct synthetic jobs using the resource profiles collected from a set of production pipelines from Microsoft’s Bing clusters. We describe the overall profile of the workflow as an RDL expression, and generate corresponding load with a synthetic time-varying job.

**Deriving SLAs** Information about SLAs are generally not available as today’s system do not provide this feature. We approach this problem as in [19]. Based on conversations with cluster operators we settle for a conservative 5% of jobs with deadlines, and a 10% “slack” (i.e., over-estimation) over the actual job resource requirements, which were known since we control job ergonomics. Deadlines are inferred as estimates from the available trace/workload information, and from conversations with job owners whenever possible. This is not ideal, but is the best we can do.

All workloads have also been scaled (by limiting max size and submission rates) to match our cluster capabilities. In the evaluation, we use a modified version of GridMix 3.0 for job submission.

## 5.2 Evaluating RDL

In this section, we provide an initial assessment of RDL expressivity and usability.

**Community feedback** We socialized RDL with over 50 practitioners from the Apache community, and extensively discussed the trade-offs between richness and usability of the language. The key ask was to keep the first iteration of the language as simple as possible. In response, we simplified the first version of RDL we released, to only allow a single-level of `all`, `any`, `order` operators in each expression (i.e., removing nesting). This limits expressivity of the language but it is likely to foster initial adoption.

**Coverage of Hive, MapReduce and Giraph** Through collaborations with the authors of [17] and [32], and several months of internal use on research clusters, we validated RDL expressivity against thousands of jobs derived from Hive, MapReduce and Giraph frameworks. In particular, we tested against over 3k Hive production queries. RDL was sufficiently expressive to capture all these jobs with sufficient resolution of details.

**History-based prediction** We conclude with an initial validation of the hypothesis that one can build good predictors from historical runs of periodic jobs. We apply a state of the art prediction algorithm<sup>7</sup> to two very large and complex production workflows from Bing clusters. The predictor’s average overestimate is well contained to 8.25%. To put this in perspective, the existing allocation mechanisms (static capacity) only allows peak provisioning. Given the large gap between peaks and average in these workflows, the baseline produces allocations equivalent to a 1470% over-estimation.

These spot results are encouraging, though we refrain from any generalization as RDL practical relevance can only be judged by real-world adoption, and production usage.

### 5.3 Evaluating Planning

In Section 3 we introduced a complete MILP formulation of our problem, and to address the MILP scalability limits we proposed two greedy algorithms GREE and GREE-L. We evaluate the *quality* of the solutions produced by MILP, GREE and GREE-L, using the following four metrics:

1. **runtime**: measured in milliseconds to place a given set of jobs. This dictates whether a solution can be used online or offline, and how well it handles large scale problems.
2. **acceptance**: measured as a weighted<sup>8</sup> count of accepted jobs. Higher degrees of acceptance are desirable as they represent higher value delivered by a cluster in terms of more guaranteed resources offered to users.
3. **preemption**: measured as the sum of vertical (down) transitions in the plan. Lower values of this metrics correspond to plans that are less likely to force the underlying system to make use of preemption (an undesirable overhead) to vacate jobs when their allocations shrink in size.
4. **uniformity**: measured as  $\frac{stddev}{average}$  of the overall plan utilization. Lower values of this metric corresponds to uniform headroom in the plan. This is correlated to lower impact of failures, and more consistent availability of resources for best-effort jobs.

**Comparing MILP and Greedy heuristics** We test MILP, GREE, and GREE-L in an offline setting (where all jobs are known in advance), focusing only on the placement aspects of our system. This allows us to perform parameter sweeps well beyond our physical cluster capacity. Our first experiment consists of placing an increasing number of atomic jobs (100-1k) randomly generated from Workload A, on a simu-

lated cluster of progressively growing size (400 to 4k nodes), while collecting measures for the metrics above. Note that the larger scenarios tested in this experiment, are the target zone for *Rayon*, i.e., large consolidated clusters. We repeat each run with several initializations and report the average of the results—all results are expressed as relative improvements over GREE.

The runtime of GREE and GREE-L range from 35 to 130ms. MILP runtime ranges from 80 to 3200 seconds (i.e., up to 5 orders of magnitude slower). The solver performance is nonetheless impressive given that the problem size exceed 250k variables and constraints. This makes MILP viable as a reference to develop heuristics but it is still not practical for online or large scale uses.

Figure 8a, shows that MILP is capable of placing more jobs than GREE, and GREE-L for small problem sizes (20 to 40% better acceptance), but the gap asymptotically disappears. We can explain this intuitively by observing that larger problem sizes, correspond to scenarios in which each job’s contribution to the overall problem is relatively small, hence the regret of a (potentially wrong) greedy decision is low. GREE-L performs close to GREE, with slightly lower acceptance rates due to its focus on lowering preemption.

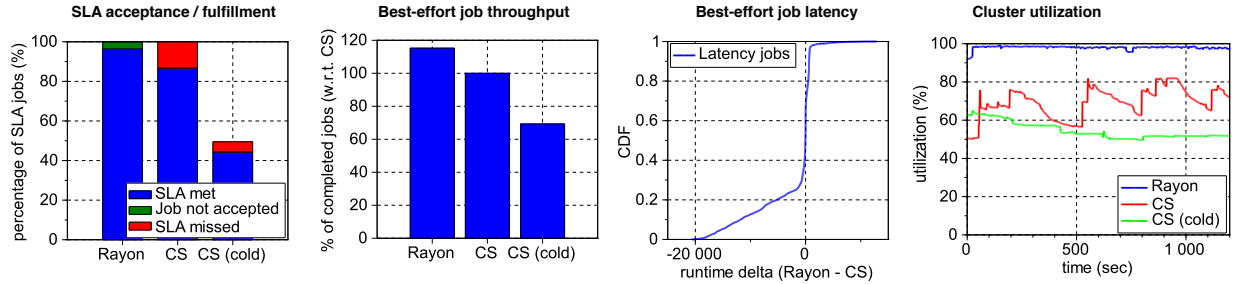
Figure 8b shows that despite accepting a larger number of jobs, MILP is capable of finding allocations with substantially less need for preemption, when compared to GREE. GREE-L on the other hand is capable of closely matching MILP preemption performance throughout the entire range. The uniformity results are similar, with GREE-L and MILP improving on GREE by 20% on average.

Comparing GREE-L and MILP on all these metrics we conclude that: **for large consolidated clusters GREE-L matches MILP solution quality, while reducing runtime by up to 5 orders of magnitude.** We started investigating hybrid strategies, that leverage MILP for the large (hard to place) jobs, and GREE-L for the bulk of small jobs. The results are inconclusive at the time of this writing.

**Impact of RDL complexity** Next we study the impact of complex RDL expressions on our placement strategies. We fix the job count to 100 but progressively change the mixture of jobs between Workload A, B, and C. Complexity of placement becomes higher, as the percentage of jobs from B and C increases. As expected, GREE and GREE-L runtimes are mostly unaffected. MILP runtimes grow sharply and hit 1 hour (an ad-hoc time bound we impose on the solver runtime) for 20% or more jobs with gang (Workload B) or 10% or more jobs with dependencies (Workload C). This is due to a drastic increase in the number of integral variables required to model the set of RDL expressions (from  $O(numJobs)$  to  $O(numJobs * timeSteps)$ ). Upon reaching this time limit the solver returns the best solution found so far (which is not optimal). For complex workloads MILP solution quality drops below GREE and GREE-L. This confirms that *MILP is a useful formal tool, but cannot scale to large or complex*

<sup>7</sup> Microsoft internal at the time of this writing.

<sup>8</sup> Weights are proportional to job size.



**Figure 9.** End-to-end experiment showing: 1) SLA acceptance and fulfillment, 2) Best-effort job throughput, 3) Best-effort job latency (as delta between approaches), and 4) cluster utilization.

problems. For these reasons we use a conservatively tuned GREE-L for all of our end-to-end experiments.

#### 5.4 End-to-end evaluation

We now turn our attention to evaluating the complete end-to-end *Rayon* architecture, running on a 256 machines cluster.

To generate a baseline, we compare *Rayon* versus the stock YARN CapacityScheduler (CS). We picked this scheduler because it is the most popular in production environments, and because we are deeply familiar with its tunables and internals [39]. Note that the relative performance we will demonstrate against YARN, would likely translate to Mesos if we were to port our ideas to that infrastructure, as Mesos also has no notion of time/deadlines, and provides instantaneous scheduling invariants very similar to YARN’s ones.

In the experiment, we generated jobs at the rate of 5,400 jobs per hour from Workload A (and later add B,C). We tuned the baseline CapacityScheduler (CS) assuming perfect workload knowledge, and manually tuning all queue parameters following industry best practices (confirmed with professional hadoop cluster operators). These means that we leverage a-posteriori knowledge on what jobs will be submitted to which queue, and assign capacity to queues optimally. What is really tested here is the best static tuning we could find, against the dynamic adaptability of *Rayon*. Details are provided in our tech report [14].

In contrast, *Rayon* configuration is basically tuning-free. We assign a maximum of 70% of the resources to production jobs, and let *Rayon*’s dynamic allocation redistribute that as needed. Best-effort jobs are assigned to queues as in the baseline. *Rayon* redistributes all unused resources among production and best-effort jobs (key to high utilization), and leverages preemption [39] to rebalance allocations.

We measure the systems under test according to the following metrics, defined over a window of time:

1. *SLA acceptance*: % of production jobs accepted;
2. *SLA fulfillment*: % of accepted jobs meeting SLAs;
3. *best-effort jobs completion*: the number of best-effort jobs processed to completion by the system;
4. *best-effort jobs latency*: completion latency for best-effort jobs;
5. *cluster utilization*: the overall resource occupancy.

Figure 9 shows the results of our experiments comparing *Rayon*, the CS, and CS running on half of the load CS(cold) according to the above metrics. CS accepts all jobs (no knowledge of deadlines) but fails to fulfill the SLA for over 15% of jobs (still non-zero when running on a fraction of the load). In contrast, *Rayon* fulfills 100% of the SLAs it accepts (more on rejection later). In the meantime, best-effort jobs throughput is increased by more than 15% and latency is improved for almost 40% of jobs. To understand why *Rayon* outperforms CS we look at rejection rates, cluster utilization, and latency for SLA jobs. *Rayon* correctly detects that not all jobs with SLA can be accepted<sup>9</sup>, and rejects about 4% of the jobs (too large to fit by their deadline). This frees up considerable resources that are leveraged to improve upon the other metrics. Moreover, *Rayon* leverages our prior work on preemption [14, 39], and thus achieves overall higher cluster utilizations<sup>10</sup>—again extra resources that we can dedicate to improve on key metrics. Finally *Rayon* leverages the fact that *SLA jobs do not care about latency* to prioritize best-effort jobs when production jobs’ deadline are not imminent. This priority inversion leads to a by-design larger latency for 60% of SLA jobs, and allows best-effort jobs to be run earlier/ faster, thus improving their latency.

These results are further analyzed in [14].

**Impact of Over-reservation** In Section 2.1, we discussed how users can define their RDL expressions. A reasonable question is “what happens to *Rayon*’s performance if users make mistakes while requesting resources?”. In case of under-reservation the answer is simple, the production job will run with guaranteed resources up to a point, and then continue as a best-effort job until completion (thus, subject to uncertainty). Over-reservation, on the other hand, affects job acceptance. To measure this we repeat the above experiment, but we systematically over-reserve by 50% (i.e., each jobs specify in its RDL expression an amount of work  $w$  that is 50% above its true needs). Due to space limitation we omit the graph, but observe the following key effects: 1) job acceptance is reduced (11% of jobs are rejected), 2) SLAs

<sup>9</sup> These rejections, albeit negative, happen at reservation time, which is much better than unannounced violation of the job’s deadline.

<sup>10</sup> In separate tests, we confirmed that preemption alone is not sufficient to fix the SLA violations of the CapacityScheduler, though it helps to increase utilization, more details in [14].

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%. Note that the drop in acceptance is less than the over-reservation. This is due to the online nature of our acceptance, and made possible by the adaptive scheduler, anticipating job and removing reservations for completed jobs.

**Handling Mixed Workloads** We validate *Rayon*'s ability to handle mix workloads, by repeating the experiments of the previous section with a mixture of 80% of Workload A, 10% of Workload B, and 10% of Workload C (our best guess of a likely mixture in consolidated clusters). This introduces jobs with gang semantics, and inter-job dependencies.

Analyzing these runs, we confirm two key hypothesis: 1) Giraph jobs from Workload B, gets access to all resources nearly instantaneously, instead of trickling of resources (as it happens with CS), and 2) *Rayon* manages to achieve high cluster utilization (near 100% after a warm-up phase) even when tasked with mix workloads<sup>11</sup>.

## 6. Related Work

While *Rayon* draws from a large body of existing work in system, and scheduling literature, the decomposition of the problem and its practical implementation are novel. To the best of our knowledge, no system handles this consolidated workload of production and best-effort jobs at high cluster utilization by explicitly managing allocations over time.

**Big-data resource management** YARN [39], Corona [2], Omega [35] and Mesos [23] invite direct comparisons to *Rayon*. While we implement *Rayon* over YARN, we believe that one could apply planning and reservation techniques to any of the prenominate systems. As of this writing, none allocate resources in time. Reservation-time planning creates opportunities for *Rayon* unavailable to online schedulers, particularly for gang requirements, workflow allocations, and admission control for time-based SLAs. *Rayon* can provide a substrate to extend invariants— such as fairness and locality [25, 43]— and techniques— such as multi-resource sharing [9, 21]— over time.

**HPC resource management** HPC schedulers [3, 36, 38] cover a complementary space, particularly for gang allocations of MPI frameworks. Where available, *Rayon* leverages fault and preemption tolerance of application frameworks to place, anticipate, and replan allocations. Parallel efforts in the big-data space make the same assumption [5, 12, 39]. Fault-tolerance is currently under review for the forthcoming 4.0 MPI standard,[4] but it cannot be assumed by HPC platforms. As a consequence, the isolation guarantees in HPC clusters are stronger, but at the expense of utilization. Grid systems like GARA [20] also use reservations to defer al-

location, but *Rayon* adds support for dependencies and supports a more abstract language.

**Deadlines and predictability** Predicting execution times and deadlines of batch frameworks [18, 19, 40] is largely complementary to *Rayon*. These systems neither provide a declarative language like RDL, nor support gangs, inter-stage dependencies, and multiple frameworks. Bazaar [27] does not consider preemption in its allocation of VMs and network resources. Lucier [29] assumes work-preserving preemption in its allocations, but dependencies are not explicitly modeled.

**Resource definition languages** Many declarative languages for resource definition are input to resource managers. One early example is IBM JCL. SLURM [41] supports a rich set of algorithms for inferring job priority and mechanisms to evict, suspend, and checkpoint jobs based on operator-configured policies. In contrast, RDL is abstract, its allocations are fungible, and are not bound to a host until a tenant generates demand. As used in GARA [20], RSVP [44] and RSL [15] specify particular resources rather than nested reservations of abstract requirements.

**Packing and Covering** Prior applications of optimization techniques to resource allocation inspired our MILP formulation. In particular, work minimizing workload makespan, [6, 22, 30], satisfying deadlines and SLAs [7, 28], and guaranteeing latency in mixed workloads [28]. These formulations explore the theoretical aspects of the problem, but do not cover all the properties of our workloads.

## 7. Concluding Remarks

Modern big-data clusters run a diverse mix of production workflows and best-effort jobs. These have inherently different scheduling needs. In this paper, we make the case for *reservation-based scheduling*, an approach that leverages explicit information about job's time-varying resource needs, and completion SLAs, to plan the cluster's agenda. We make four key contributions: 1) a declarative reservation definition language (RDL) to capture such information, 2) a framework for planning where we characterize the scheduling problem using an MILP formulation and develop fast, greedy heuristics, 3) adaptive scheduling of cluster resources that follows the plan while adapting to changing conditions, and 4) a system, named *Rayon*, that integrates the above ideas in a complete architecture. We have implemented *Rayon* as an extension to Apache YARN and have released it as open-source. Our experimental results confirm that temporal planning of the cluster's agenda enables *Rayon* to meet production SLAs, while providing low-latency to best-effort jobs, and maintaining high-cluster utilization.

*Rayon* is our first step towards addressing this problem. Ongoing work in collaboration with the authors of [17], and [32], is geared towards addressing *Rayon*'s usability. We are also exploring more sophisticated planning algorithms, and economy-based models for resource reservation.

<sup>11</sup> We visualize all of this further in [14].

## References

- [1] Apache Giraph Project. <http://giraph.apache.org/>.
- [2] Facebook Corona. <http://tinyurl.com/fbcorona>.
- [3] Maui Scheduler Open Cluster Software. <http://mauischeduler.sourceforge.net/>.
- [4] MPI 4.0: fault tolerance working group. <https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/FaultToleranceWikiPage>.
- [5] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True Elasticity in Multi-Tenant Clusters through Amoeba. In *SoCC*, October 2012.
- [6] J. Augustine, S. Banerjee, and S. Irani. Strip packing with precedence constraints and strip packing with release times. 2006.
- [7] A. Balmin, K. Hildrum, V. Nagarajan, and J. Wolf. Malleable scheduling for flows of mapreduce jobs. Technical report, Research Report RC25364, IBM Research, 2013.
- [8] K. Bellare, C. Curino, A. Machanavajihala, P. Mika, M. Rahurkar, and A. Sane. Woo: A scalable and multi-tenant platform for continuous knowledge base synthesis. *PVLDB*.
- [9] Bhattacharya, C. Arka, F. David Culler, G. Eric Friedman, S. Ali, a. S. Scott, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *SoCC*, 2013.
- [10] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads. *PVLDB*, 2012.
- [11] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *MASCOTS*, 2011.
- [12] 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. 2013.
- [13] B.-G. Chun. Deconstructing production mapreduce workloads. In *Seminar at: http://bit.ly/1jZOPgT*, 2012.
- [14] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: if you're late don't blame us! (extended version). In *Microsoft Tech-Report: MSR-TR-2013-108*, 2013.
- [15] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Job Scheduling Strategies for Parallel Processing*. Springer, 1998.
- [16] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [17] A. Desai, K. Rajan, and K. Vaswani. Critical path based performance models for distributed queries. In *Microsoft Tech-Report: MSR-TR-2012-121*, 2012.
- [18] X. Dong, Y. Wang, and H. Liao. Scheduling mixed real-time and non-real-time applications in mapreduce environment. In *ICPADS*, 2011.
- [19] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *EuroSys*, 2012.
- [20] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Quality of Service, 1999. IWQoS'99. 1999 Seventh International Workshop on*. IEEE, 1999.
- [21] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *NSDI*, 2011.
- [22] E. Günther, F. G. König, and N. Megow. Scheduling and packing malleable and parallel tasks with precedence constraints of bounded width. *Journal of Combinatorial Optimization*, 2014.
- [23] B. Hindman and et al. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.
- [24] S. Ibrahim, H. Jin, L. Lu, L. Qi, S. Wu, and X. Shi. Evaluating mapreduce on virtual machines: The hadoop case. In *Cloud Computing*, pages 519–528. Springer, 2009.
- [25] 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.
- [26] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur. Oozie: towards a scalable workflow management system for hadoop. In *SWEET Workshop*, 2012.
- [27] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *SoCC*, 2012.
- [28] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula. Calendaring for wide area networks. *SIGCOMM*, 2014.
- [29] B. Lucier, I. Menache, J. S. Naor, and J. Yaniv. Efficient online scheduling for deadline-sensitive jobs: extended abstract. In *SPAA*, 2013.
- [30] K. Makarychev and D. Panigrahi. Precedence-constrained scheduling of malleable jobs with preemption. *arXiv preprint arXiv:1404.6850*, 2014.
- [31] V. Mayer-Schönberger and K. Cukier. *Big data: A revolution that will transform how we live, work, and think*. Houghton Mifflin Harcourt, 2013.
- [32] A. D. Popescu, A. Balmin, V. Ercegovic, and A. Ailamaki. Towards predicting the runtime of iterative analytics with predict. Technical report, EPFL, 2013.
- [33] D. Sarkar. Introducing hdinsight. In *Pro Microsoft HDInsight*, pages 1–12. Springer, 2014.
- [34] 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.
- [35] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.
- [36] G. Staples. Torque resource manager. In *IEEE SC*, 2006.
- [37] R. Sumbaly, J. Kreps, and S. Shah. The big data ecosystem at linkedin. In *SIGMOD*, 2013.

- [38] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor: a distributed job scheduler. In *Beowulf cluster computing with Linux*, 2001.
- [39] V. Vavilapalli and et al. Apache hadoop yarn: Yet another resource negotiator. In *SoCC*, 2013.
- [40] A. Verma, L. Cherkasova, and R. H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *ICAC '11*, 2011.
- [41] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, 2003.
- [42] N. E. Young. Sequential and parallel algorithms for mixed packing and covering. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 538–546. IEEE, 2001.
- [43] 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*, 2010.
- [44] L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (rsvp)–version 1 functional specification. *Resource*, 1997.