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

Carlo Curino[m], Djellel E. Difallah[u], Chris Douglas[m], Subru Krishnan[m],
Raghu Ramakrishnan[m], Sriram Rao[m]
Cloud Information Services Lab (CISL) — Microsoft Corp.
[m] : {ccurino, cdoug, subru, raghu, sriramra}@microsoft.com, [u]: djelleleddine.difallah@unifr.ch

Microsoft Tech-Report: MSR-TR-2013-108

ABSTRACT

In recent years, organizations have increasingly shifted toward a data-driven approach to business—this led to an explosion in the complexity and variety of big-data applications. Furthermore, we are observing a progressive shift from clusters dedicated to a single purpose (e.g., production MapReduce jobs) to large, consolidated clusters running a mixture of production and testing applications.

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

We propose a *resource description language* that allows users to specify their resource needs abstractly, 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 request. This allows large production pipelines with deadlines to share a cluster with small, ad-hoc, latency-critical jobs. We show the power of this approach by presenting a scheduling framework that uses this rich language to ensure *predictable resource allocation for production jobs while minimizing latency for best-effort jobs*. Our framework relies on admission control, work-preserving preemption and quick adaptation to changing cluster conditions to achieve all of this, without sacrificing utilization.

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 a popular scheduler. We present extensive experimental evaluation by running our system on a 256-node cluster using ten workloads derived from real-world traces from clusters of Cloudera customers, Facebook, Microsoft, and Yahoo!.

1. INTRODUCTION

Over the past decade, scale-out computing over large clusters 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. We highlight two important trends in this industry.

Cluster Consolidation: There are early signs of a fundamental shift in datacenter cluster workloads: clusters which were traditionally single-purpose (e.g., dedicated to a production pipeline or to testing/research), are now being consolidated/shared. This aims to increasing hardware utilization and thus improve return on investment (ROI) [26]. This is akin to the general benefits of the cloud. In this new world, recurrent, production pipelines, run in a shared environment together with a large number of small, best-effort, jobs—such as interactive queries, or small-scale debug runs.

Workload Diversity: The types of computations people run have diversified from MapReduce jobs to interactive analytics, stream and graph processing, iterative machine learning, MPI-style computations [11, 8, 9], and complex pipelines (DAGs of jobs) leveraging multiple frameworks [5]. This imposes novel requirements on scheduling infrastructures which were solely targeted to malleable batch jobs [18, 26, 29].

Our hunch is that both *consolidation and diversity are coped with today in cumbersome and costly ways, involving a great deal of over-provisioning and manual intervention*. We confirm this intuition, by talking with several cluster operators and practitioners, both inside and outside Microsoft. The war stories we gathered denounce shortcomings of currently deployed systems, that we believe are not fully addressed even by recent literature. Common side-effects include: 1) frequent meetings between cluster operators and key users dedicated to planning job submission times, and resource allocations to ensure proper execution of business-critical workloads— this was jokingly referred to as *"arm-chair scheduling"*; 2) need for personnel dedicated to guarantee timely progress of production pipelines, by *manually killing* best-effort jobs when resources become scarce; 3) resource under-utilization caused by peak-provisioning for complex pipelines; 4) hoarding of resources (which wastes resources by idling them), to support gang-oriented scheduling atop batch-oriented schedulers; 5) (despite the above) constant *user complaints of uncertainty* about job runtimes.

Modern resource managers such as Mesos [18], Omega [26], Hadoop YARN [29], and Corona [2] provide a necessary first step towards supporting consolidation and diversity by lifting programming model restrictions, boosting scalability, and providing extensible architectures —see Figure 1. The key remaining limitation is: *lack of predictability and time-awareness*. The culprit is an historically well-justified bias towards time-oblivious mechanisms provided by the underlying scheduling infrastructures (e.g., instantaneous fairness [18, 2], fixed priorities [26], static capacity partitioning [29]), which forces users to compensate as we discuss above

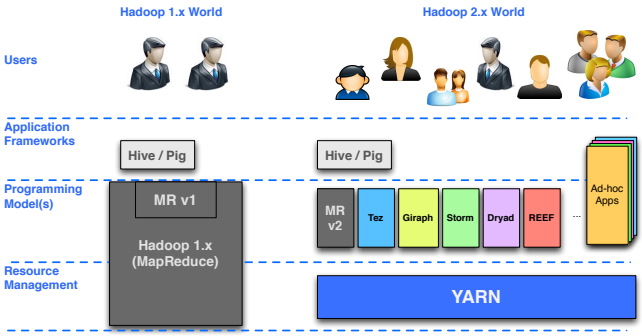

Figure 1: Evolution of big-data frameworks: the Hadoop ecosystem.

(§ 2). 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 relationship between users and operators exacerbate these problems.

Our contributions: the key intuition behind our system is to introduce a layer of resource reservation and planning. Consequently, by introducing admission control and careful resource budgeting we enhance existing scheduling infrastructures to *support diverse application frameworks in sharing a highly utilized cluster, guaranteeing SLAs for production jobs, and minimizing latency for best-effort jobs* (§ 3).

To this purpose, we allow users (or tools on their behalf) to specify their resource requirements ahead of job execution by means of a rich Resource Description Language (RDL) that can capture both malleable and gang-scheduling (or simply, *gang*) allocation needs, time requirements, and dependencies among stages of a pipeline. We then extend modern big-data systems to understand and leverage this information to deliver *predictable resource allocation*. In a nutshell, by supporting the RDL language we allow users to expose to the system their true resource requirements, and obtain immediate feedback about when their job will be executed. The scheduler uses this additional information to plan the cluster's agenda: by distinguishing between production jobs that need to be executed by a certain deadline, from latency sensitive jobs submitted interactively by users our system can prioritize resource allocations in a time-dependent manner. Effectively, this is akin to enhancing big-data resource managers, which excel at fine-grained sharing, scalability, and fault-tolerance, with admission control, and time-aware scheduling features which have been the focus in HPC settings.

The conceptual contributions we propose can be applied equally well to systems like Mesos, Omega, Hadoop YARN, and Corona. Of these systems, for practical reasons, we chose Hadoop YARN as the starting point to build our system (§ 4). The resulting framework, named *Rayon*, is being contributed to Apache Hadoop[1].

This paper is mostly about *Rayon* as an infrastructure. However, in order to compare with the state of the art, we implemented simple placement and replanning policies, and pitch our system against a production-class scheduler in an extensive experimental evaluation based on workloads derived from Cloudera, Facebook, Linkedin, Microsoft and Yahoo! production clusters(§ 5). Comparing *Rayon*, even with simple heuristics, against YARN's main scheduler we observe: 1) an almost tuning-knob free experience, 2) 0% SLA violations (vs 15% for the baseline) while accepting over 96% of SLA jobs, 3) latency is reduced for almost 40% of best-effort jobs.

[1]We will track progress at: `https://issues.apache.org/jira/browse/YARN-1051`.

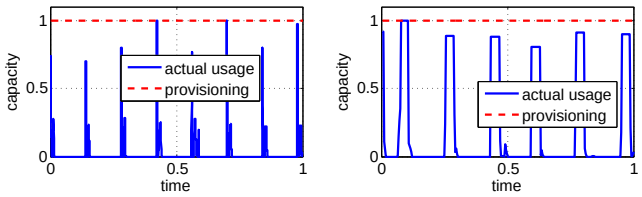

Figure 2: Two Microsoft production pipelines.

4) 30 to 50% increase in cluster utilization, and 15% increase in cluster throughput.

The ultimate goal of *Rayon*, is to bridge the gap between a vast literature of sophisticated and optimal algorithms for scheduling and resource management, and the stark reality of currently deployed systems (§ 6). Our extensible design, and open-sourced Hadoop implementation provides an ideal substrate for research exploration, by lowering the barrier of entry (all mechanisms are provided), and maximizes chances of real-world impact. This is confirmed by a great deal of excitement we gathered when discussing *Rayon* with our peers. Among the many that expressed interest, we are actively collaborating with three groups of researchers to adapt and extend their efforts in optimal placement [22], incentive design, and resource profiling [12] to *Rayon*. On the other hand, we are fostering adoption by engaging with key Hadoop developers and users, and evolving *Rayon* to incorporate their needs (§ 7).

2. STATE OF AFFAIRS

In this section, we summarize some salient characteristics of modern big-data workloads, by combining public information that appeared in recent reports [11, 8, 9, 14, 5] and direct experience of Yahoo!, Microsoft, and Quantcast clusters. We use this to derive high-level requirements.

In first approximation, all jobs running in a cluster can be classified in a coarse but useful way as:

1. *Best-effort jobs*: These are ad-hoc, exploratory jobs submitted by data scientists and engineers engaged in testing/debugging of ideas. They are typically small (running on synthetic or sampled data) and interactive. Latency is a key concern.
2. *Production jobs*: these (DAGs of) jobs are submitted recurrently by automated systems, such as Oozie [20]. They are typically much larger in size, and associated with stringent SLAs, such as completion deadlines.

Clusters have drastically different mixtures of best-effort and production jobs [29, 11, 8, 9], yet the surprising invariant is that even when outnumbered, production jobs take the lion's share of the resources—above 95% in all but a few test clusters. It is also very common for big-data clusters to run complex pipelines, and jobs with gang requirements, i.e., jobs that require all of their tasks to run simultaneously.

Talking with several cluster operators, we gather that consolidating these workloads is very appealing to increase ROI, but it is quite challenging given the available scheduling infrastructures. All popular big-data systems today [18, 26, 29, 2], provide only time-oblivious mechanisms (queues/pools/priorities) to share cluster resources. Consequently, mapping time-varying phenomenon in terms of static notions of capacity, fairness, or priority, imposes an uncomfortable trade-off between consistency in satisfying user expectations (latency and deadlines) and cluster utilization. For instance, peak-provisioning and static partitioning of resources will satisfy the users by delivering very predictable executions but at a very high cost, while high-utilization and soft-boundaries can lower costs, by hoping for acceptable average behaviors.

For completely malleable jobs such as MapReduce the situation is dire but bearable, as the job's natural flexibility and fault-tolerance can be leveraged to find usable compromises. Things get unsustainable when trying to consolidate workloads that include big production pipelines or jobs with gang-requirements—an unfortunately common scenario. We show why this is problematic by: 1) analyzing the over-provisioning needs of two Microsoft production pipelines, and 2) running Giraph (gang requirements) with and without dedicated resources.

Pipelines are characterized by extreme peak-to-average ratios [14, 5], which combine poorly with the static resource allocations mechanisms. To guarantee that SLAs will be met cluster operators are forced to peak-provision. This means that vast amount of resources are dedicated to a pipeline, even though they are not needed most of the time. This is shown in Figure 2 for two very large production pipelines from Microsoft clusters, which show high peaks separated by long periods of low utilization. The unused resources can be tentatively allocated to run other jobs, but with very little promises in terms of predictability. The only alternative is to leave resources fallow. Neither is satisfactory for large production pipelines.

To further investigate some of the difficulties of consolidating modern workloads, we run a simple PageRank computation using Apache Giraph [1] on top of YARN. Figure 3 shows that when resources are statically provisioned for this job, their acquisition by the job happens quickly and execution runtime is low and predictable (dashed-line on left). The same figure also shows the results of a Giraph without dedicated resources and in the presence of another job. When these jobs are run concurrently, since Giraph does not have dedicated allocation, it is forced to "hoard" resources to fulfill its gang requirement, only at this point the actual computation can start. This leads to: 1) increased and unpredictable runtimes, 2) wasted resources during hoarding (grayed-out area), and 3) risk of deadlocks among multiple gang jobs (not shown). The magnitude of these effects increases as clusters get busier.

Handling such mixed workloads today requires significant over-provisioning and frequent manual intervention. This is already problematic for on-premise clusters and will be exacerbated by public-cloud, big-data-as-a-service environments, where small groups of internal users are replaced by a crowd of paying customers, and informal expectations evolve into contractual obligations.

All of the above points to the need for a framework capable of supporting:

1. **complex workloads**: handling flexible and gang jobs, and arbitrary pipelines with inter-stage dependencies;
2. **production SLAs**: predictable execution and completion times for production jobs;
3. **best-effort jobs latency**: optimized scheduling to lower latency for interactive, ad-hoc jobs;
4. **cluster throughput**: dense packing of jobs to increase utilization, and thus ROI.

In Section 6, we discuss how the extensive prior work [29, 18, 26, 2, 28, 3, 27, 30, 14, 13, 22, 21, 31] has tackled different subsets of this problem, and how many practical workaround have been attempted. To the best of our knowledge, *Rayon* is the first system to provide a practical infrastructure designed to address all of the above. We present *Rayon*'s architecture next.

3. THE RAYON FRAMEWORK

The goal of our system is to deliver SLAs for complex production jobs (with deadlines, gang-requirement, and pipelines), low latency for best-effort jobs, and maximize cluster utilization. We

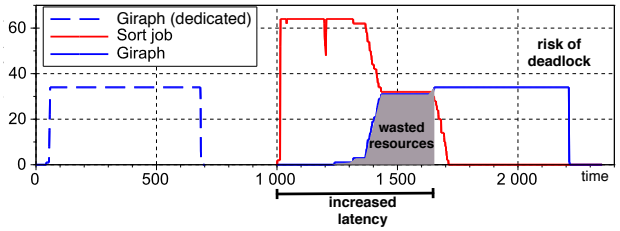

Figure 3: Shortcoming of running gang jobs, without guaranteed resources.

begin by providing an overview of our system (§ 3.1), then describe the RDL language (§ 3.2), resource management issues (§ 3.3), and conclude this section discussing the resource definition problem (§ 3.4).

3.1 Overview

Modern big-data frameworks (such as, Mesos, Omega, YARN, and Corona) have introduced a crisp separation between a per-application *job manager*, in charge of supporting a specific programming model, and controlling the execution flow of a single job, and a *resource manager*, an underlying piece of infrastructure handling scheduling, and multi-tenancy.

Figure 4 shows how we embrace this architectural pattern and decompose the problem of providing SLAs in a shared environment into two sub-problems: 1) *resource definition*: i.e., determining the time constraints and resource needs of a job, and 2) *predictable resource allocation*: i.e., the problem of determining whether a job can be admitted, when and how it should be run, and executing on such plan coping with the complex dynamics of a multi-tenanted cluster. The first problem is mainly dependent on the given job (e.g., its parallelism, code efficiency, data skew), and we argue that it is the responsibility of the *job manager*, while the second is a scheduling/planning problem that requires cluster-state visibility, and consideration of other jobs' needs. This should be handled by the *resource manager*. The primary focus of this paper is on handling the predictable resource allocation problem (§ 3.3). For completeness, we also provide an overview of how one can tackle the resource definition problem (§ 3.4).

Given this architecture we need a crisp and flexible way to describe a job's resource needs and time constraints. 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-stage dependencies of a pipeline). The latter dimension, in particular, is not addressed satisfactorily by current systems (see § 2), and represents a key extension that we propose.

The job manager relies upon the resource manager to reliably satisfy resource requests expressed in RDL, and in turn exposes a great deal of flexibility. Specifically, by representing the job needs in a declarative way, we enable the resource manager to leverage information about jobs' ergonomics and time constraints, to provide predictability without compromising on cluster utilization.

Our design of the resource manager, shown in Figure 4, consists of two parts: the *planner*, which leverages pluggable policies to organize the cluster's agenda (i.e., which jobs are to be run, when, and with what resources), and the *allocator*, which exploits work-preserving preemption [10, 29] and leverages existing scheduling infrastructures as a building block to elastically grant and revoke

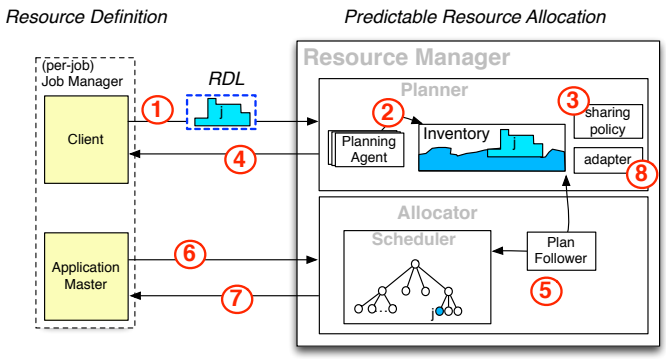

Figure 4: Overview of the *Rayon* system.

access to resources based on the planner's indications.

The job manager estimates the demand generated by a job or pipeline and encodes its constraints as a *request expression* in the *Resource Description Language* (RDL) defined in § 3.2. This definition step is not the focus of our paper, but we provide an informal survey of techniques that can be leveraged to produce these estimates (§ 3.4).

Given an RDL request expression generated by a job manager, we next describe how *production jobs* are admitted by the system. The sequence of steps outlined in Figure 4 are as follows:

Step 1 The resource request is submitted to the planner by a component of the job manager in charge of the initial negotiation and job submission, namely, the Client.

Step 2 The planner maintains an *inventory* of the commitments made on cluster resources by tracking all reservations. The planner leverages a set of pluggable *planning agents*, to resolve an RDL request against the inventory. Different agents can be used to bias which of the many valid solutions should be picked (e.g., the one minimizing latency, preemption, avoiding busy periods).

Step 3 The resulting allocation is validated both against physical resource constraints. and sharing policies. By way of illustration, a sharing policy could enforce time-extended notions of user quotas. We describe how the planner makes this determination in § 3.3.1.

Step 4 Immediate feedback is provided to the user[2] on whether the reservation succeeded or failed, and optionally how it was satisfied. Note that this reservation process can happen significantly ahead of time, with respect to the actual job/pipeline execution. Once a RDL request is accepted and accounted for in the inventory, it defines a *contract* between the user and the system. The expectation is that the system under all but extreme conditions will fulfill this contract, by providing a predictable resource allocation.

Step 5 The allocator leverages the competence of current scheduling infrastructures in enforcing instantaneous invariants and adapting to changes in cluster state. A new component, the PlanFollower, affects the scheduler behavior by continuously updating the scheduler target invariants according to the inventory (step 4 in Figure 4). This two level view of the problem (inventory vs scheduler) is practically motivated, as the inventory maintains aggregate view of resources ignoring details of task and node placement, but has full details of the time dimension, while the scheduler is only concerned with the current allocations, but deals with the full spatial complexity of tasks and nodes in the cluster. Each components complexity and run-time load is thus kept manageable.

Step 6 When the job is finally started, the runtime component of the job manager, the *Application Master*, will request resources based on instantaneous needs from the job. These requests are always fulfilled within the limits of accepted reservations, and are satisfied on a best-effort basis if they exceed the reservation. This makes for a forgiving system, and fosters high cluster utilization.

Step 7 The job manager receives access to resources and proceeds to spawn the tasks of the job accordingly. This aspect is described in § 3.3.2.

Step 8 The *adapter* is a component that makes *Rayon* responsive and dynamic. The adapter dynamically rearrange the inventory in response to changes in cluster capacity (node failures or additions), or simply to improve the allocation by applying off-line scheduling techniques. Similarly, the job-manager might detect that 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, subject to the same validation process.

We presented this flow as if each RDL reservation is associated with a single job, but more generally each single reservation can support at runtime an arbitrary number of jobs—we discuss the advantages of this choice in Section 3.4. In order to maintain high cluster utilization, the allocator components very aggressively redistributed both unreserved, and reserved but unused resources among the all jobs in the system according to existing fairness/capacity semantics [16, 29]. By leveraging work-preserving preemption [10, 29], this can be done without delaying rightful owner of resources when they show up, nor wasting progress of jobs running on borrowed resources when preempted.

This mechanism is leveraged also to support *best-effort jobs*. These jobs are modeled as jobs with an empty reservation, i.e., they have no "guaranteed" access to resources, but will be given resources opportunistically. Since no deadline is specified resources are provided as soon as possible, thus aiming at minimizing latency.

We conclude this overview by pointing out that most of the policies we mentioned are currently simple heuristics aimed at demonstrating the soundness of our design, and *Rayon*'s potential vs state of the art. Exploring algorithms for optimal on-line placement, off-line replanning, dynamic adaptation, resource definition, etc. is part of our ongoing research.

Next we present the RDL language.

3.2 Resource Description Language

An RDL *request expression* is one of the following types of expressions:

1. An *atomic expression* of the form $\langle g, h, t, w \rangle$, where: g and h are the minimum and maximum parallelism for the step; a valid allocation of capacity at a time quanta is either 0 or in the range $[g, h]$, t is minimum duration of consecutive allocations; a valid allocation persists for at least t steps, and w is the threshold of work necessary to complete the reservation; the expression is satisfied iff the sum of all its allocations is w. In Figure 5a, we show an example of atomic expression.
2. 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.

[2]This makes our problem an online scheduling one.

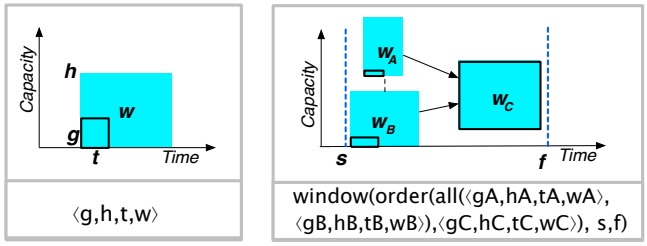

Figure 5: Two RDL expressions

3. 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.
4. A *dependency expression* of the form $order\{e_1, e_2, \ldots e_n\}$. It is satisfied if for all values of i the expression e_i is satisfied with allocations that strictly precede all allocations for e_{i+1}.
5. 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.

It is also 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*, the *any* operator allows a job manager to propose multiple physical plans for a single logical plan, granting the planner significant flexibility. Finally, the *window* operator can constrain a job to receive resources only during a particular interval, which is ideal for expressing deadlines.

Note that all request expressions typically admit multiple solutions for a given inventory. Choosing between equivalently valid allocations is a prerogative of the planning agents. Different planning agents can be used to encode job-specific "preferences", e.g., minimize preemption, lower latency, etc. This customizability of planning agents is the key to *Rayon*'s extensibility.

So far we referred to capacity appealing to the reader's intuitive understanding of this concept. We can define it more precisely as follows: *a unit of capacity in RDL is a multi-dimensional bundle of resources $\langle RAM, CPU, \ldots\rangle$, with an associated locality preference*. E.g., I need 10 bundles of $\langle 2GB, 2cores, \ldots\rangle$ on Rack2. This is closely related to YARN's notion of containers [29]. Parallelism is then defined in terms of these units. In Section 4, we discuss two practical simplifications we make in our prototype, to help *Rayon* scale, and handle fast-evolving cluster conditions.

While we do not make completeness claims about the RDL language, we find it sufficient to naturally capture a broad class of practical scenarios. We are validating this by socializing our design with the Apache community, as part of *Rayon*'s open-sourcing.

We next provide an example of how a job manager encodes its requirements in RDL.

3.2.1 Example: A Recurring pipeline

For purposes of illustration, as shown in Figure 5 (b), we model a recurring workflow of three jobs in which two are malleable batch jobs $\{A, B\}$ (e.g., MapReduce), and one $\{C\}$ has a gang requirement (e.g., Giraph). The input is known to become available at time s and the production pipeline has a completion deadline of f. Assume that C depends on the output of both A and B. Since our temporal constraints are on the entire pipeline, we place a $window$ expression at the top level, $e = window\{e', s, f\}$. Next, to describe our DAG dependencies e', note that (1) A and B can run concurrently, and (2) C can start only after A, B finish execution. This can be expressed in RDL as $e' = order\{all\{e_A, e_B\}, e_C\}$. Hence, the expression that describes all the temporal constraints for this pipeline is: $e = window\{order\{all\{e_A, e_B\}, e_C\}, s, f\}$. Figure 5 (b) shows a visual representation of e including the resource definitions for various stages of the pipeline.

Next we turn our attention to defining the atomic expression for each job: $\{e_A, e_b, e_C\}$ The first two jobs are malleable so g_A and g_B are set to 1, and h_A and h_B to the maximum number of tasks that can be run in parallel (e.g., max of number of maps and reduces). C has gang requirements so $g_C = h_C$ and it is set to the exact number of tasks that the job needs to run. Similarly t_A and t_B are set to the expected run-time of each task (e.g., 95th percentile of expected maps duration) to guarantee that most tasks can complete within their guaranteed allocation, while t_C correspond to the overall duration of the job (gang semantics over time). The values of w_A, w_B, w_C are derived from the above, and the overall expected computation required by each job.

3.3 Resource Manager

The resource manager determines whether a request expression can be accepted without violating physical constraints, overlapping existing contracts, and respecting sharing policies. Once accepted, an RDL request becomes a contract, which will be honored at runtime, by providing at least the promised resources to jobs that are submitted within it. This problem is referred to in literature as *commitment on arrival* [22] 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 backfilling 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 (§ 3.3.1), allocator (§ 3.3.2), and adapter (§ 3.3.3) at a conceptual level. We conclude the resource manager design with a discussion of policies for quota management (§ 3.3.4).

3.3.1 Planner

On receipt of a request expression e generated by a job manager, the planner invokes a planning agent to find a valid allocation that satisfies e. Note that there maybe multiple planning agents that can yield a valid allocation for e. The job manager can influence this allocation choice by picking one of the available agents (or as an extension by providing its own planning agent). The underlying placement problem has inherently bad complexity properties. More formally we can state that:

THEOREM 3.1. *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 [15] 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 NP-completeness result, prior impossibility results [22] on designing optimal algorithms for commitment on arrival scheduling problems, and the practical considerations owing to fast-evolving conditions in large clusters requires us to focus on robust heuristics in our implementation. In particular, we explore in § 4 a series of greedy heuristics that can find solutions in linear time for a practically important subset of RDL expressions.

3.3.2 Allocator

A job accepted by the planner has entries in the inventory describing its logical reservations in a given time interval. When that interval arrives, the *allocator* uses the inventory to provide the job with physical resource allocations to satisfy its logical reservations.

The allocator simply follows the logical plan generated by the planner and runs a simple algorithm shown in Algorithm 1. By simply iterating over the plan, we can satisfy all our guarantees in that interval. Any resources that are left-over (i.e.,"idle" resources) can now be distributed to the jobs according to whatever policy—fairness, capacity, priority, etc—and thereby improve cluster utilization.

Algorithm 1: 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)  a ← ∅
d_r ← Σ 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 − Σ inv[t])
def assign(a : Alloc, (P) : Policy, J : Job[], c : Alloc)
    // assign remaining resources to jobs
     using
    // fairness, capacity, or other metrics
def scale((P) : Policy, c : Alloc, inv : Inv)
     // reassigns committed resources
   // according to policy function
```

3.3.3 Adapter

Run-time considerations may cause the job manager and the resource manager to modify the reservations committed to the inventory by the planner. 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 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.

The job manager may also need to reevaluate its commitments if there is a significant fluctuation in when the job completes relative to plan. For instance, jobs may complete before their reservation expires, 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. To address all of these run-time issues, we introduce an *adapter* component which replans the inventory so that it can tightly pack these reservations. Similar mechanisms can be leverage to respond to fluctuations in the overall cluster capacity, and they amount to off-line scheduling. This is another important extensibility point for *Rayon*. In Section 4, we discuss the current heuristics we leverage for replanning.

3.3.4 Quota management

Rayon's inventory requires a sharing policy to handle multi-tenancy aspects and in particular, to prevent users from monopolizing cluster resources. This problem is related to incentive design and it is a complex problem on its own right. While this policy component in *Rayon* is pluggable, we describe an initial policy that extends the notion of capacity sharing over the time domain (i.e., quotas-over-time). This is a generalization of the notion of capacity sharing from YARN [29].

The key idea is that instead of enforcing capacity instantaneously as done by existing schedulers, we enforce an average capacity C over a window of time win, e.g., an average of 10% of cluster resources every 24 hours. This allows user to obtain reservation that exceed the 10% limit, for a short period of time, but ensure that the quota-over-time is not violated. This policy is invoked once a specific allocation a is found for an RDL expression, but before it is accepted. The validation step checks that the sum of all resources reserved by this user/group, including a, does not exceed $C * win$ in any window of size win that a overlaps with. Since it is applied to each submission, this guarantees that the resource consumption of a user/group as a whole never exceeds the value of C over any contiguous windows of length win.

This provides a new and interesting trade-off between flexibility of allocation and fairness. Which administrators can explore by setting the values of C and win—in particular if win is set to the unit value of 1, this policy correspond to standard instantaneous capacity.

3.4 The resource definition problem

Recall that *not all jobs need to be expressed in RDL*. While RDL is a powerful tool to expose the requirements of production jobs, our system naturally supports the execution of best-effort jobs side-by-side to production ones, and according to popular scheduling semantics [29, 32, 16, 6].

For production jobs, our approach is postulated on users (or automated tools) ability to express their workloads in terms of RDL expressions. While a full investigation of this sub-problem is part of our future research agenda, we briefly summarize promising angles to tackle this. Production jobs often fall into one of the following special groups:

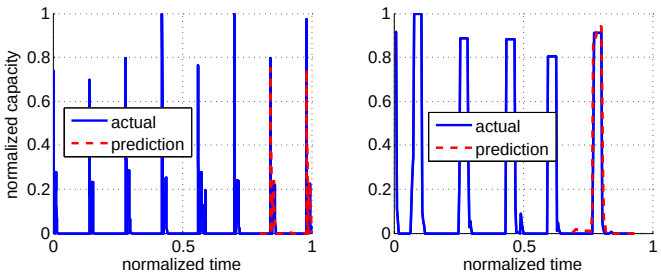

Figure 6: Predicting recurring pipelines.

High-level frameworks: Despite the diversity of workloads a significant fraction of production jobs are generated by a handful of frameworks such as Hive, Pig [11, 8, 9]. This gives us an opportunity to build profilers and optimizers capable of automatically producing precise resource demands for queries/jobs by leveraging the application semantics [30, 14, 12, 21, 13]. We are currently collaborating with the authors of [12] to explore integration of their approach within the *Rayon* framework.

Carefully analyzed jobs: Among jobs with strict SLAs the class of machine learning jobs stands out. These are are crucial to spam filtering, advertisement, user modeling. We gather anectodically that data scientists routinely perform careful analysis of their data and iterative algorithms to secure sufficient resources to meet their deadlines—engaging in what we referred to § 1 as arm-chair scheduling. In particular, given short test runs, scientists can precisely determine the required capacity, and the duration of each iteration of their algorithm. It is also commonplace to run a bounded number of iterations, to obtain a safe over-estimate of the expected algorithmic convergence. This gives a precise upper bound to the job runtime. We validate these assumptions with a small user study, where we asked ML practitioners to estimate using RDL their resource needs when running an MPI-based batch-gradient descent algorithm [23]. We tested under six previously unseen combinations for algorithm parameterization and data sizes. Given small test runs (on sample of the data, and for just few iterations), the practitioners were 100% accurate in determining the capacity required for the full scale runs, and could estimate the iteration cost within 5% in average. Algorithmic convergence was always well within the iteration bound (with a consistent 60% margin). They confirmed this is common practice, and while tighter bounds can easily be determined, for important production runs people are likely to maintain safe over-estimates to guarantee convergence. While this is not our focus, such consistency of overestimates lends itself to overcommitting of the inventory.

Periodic and predictable jobs: As we mentioned in § 2 production jobs are submitted automatically and periodically [14, 11]. This makes them amenable to history-based resource prediction [25]. We informally validate the last statement by running seasonality detection and building predictors for the pipelines from internal Microsoft clusters. In Figure 6, we show the result of applying a state of the art prediction algorithm[3] to the two pipelines we showed in Section 2. The precise matching which is visually obvious in Figure 6, can be quantified by measuring the capacity actually required versus allocated. The predictor's average over-estimation across the various stages of the pipeline is 8.25%. To put this in perspective, existing mechanisms that simply peak provision (based on 99th percentile) for the entire pipeline lead to a 1470% over-estimation! These results are promising, though we refrain from any generalization. However, we remain optimistic since the large gap between our approach and the baseline can aborb even a significant drop in prediction accuracy. We further investigate the impact of over-estimation in § 5.4.

Services and time-evolving manual provisioning: A special case of periodic reservations are services. These are conceptually long-running jobs with resource needs that fluctuate based on the external traffic they are serving. One such example is Linkedin's stream-processing system Samza [4]. We gathered that these systems are typically peak-provisioned today, but that the predictable ebbs and flows of their input traffic, can be translated naturally in time-evolving RDL requests. Similarly RDL can be used to specify time-evolving allocations to groups of users, akin to a queue with time-dependent capacity. This hints at a manual use of RDL as a flexible provisioning tool. We are exploring this further within the open-sourcing process we are engaged in.

All of above amounts to an informal but promising validation, that users or systems on their behalf can indeed leverage RDL effectively to express their needs.

4. IMPLEMENTING RAYON

We use the Apache Hadoop 2.1.0 / YARN [29], an extensive redesign of earlier versions of Hadoop, as the starting point for implementing *Rayon*.

This new architecture adopts the division between job manager and resource manager, as we described in Section 3.1. The job manager side is arbitrary user code (or more often a popular framework), and is organized in a Client handling submission and an ApplicationMaster, in charge of runtime application workflow. All scheduling decision are performed by a central component called the ResourceManager (RM), while per-node daemons, the NodeManagers (NM), provide resource monitoring and enforcement of the scheduling decisions. The YARN RM provides 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 [29]. We chose this as the underlying scheduler for *Rayon*.

The CapacityScheduler enables sharing of cluster resources through a notion of instantaneous 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.

4.1 Implementing Rayon in YARN

Our implementation involves several changes to the YARN RM which collectively enable predictable resource allocation to production jobs. We introduce a new public API in YARN, accessible by both Client(s) and ApplicationMasters, that allows applications to create/update/delete reservations expressed in terms of RDL. We map RDL concepts to existing constructs as much as possible (e.g., capacity is mapped to container requests), and introduce extensions of this constructs whenever necessary (e.g., to capture time, gang, and stage dependencies). In what follows, we describe the implementation of the various components of the resource manager (that we outlined in § 3.3).

[3]Microsoft internal at the time of this writing.

[4]See `http://incubator.apache.org/projects/samza.html`.

Planner. this API is supported by a new service running inside the RM. Each invocation of the service triggers the run of a planning agent against the inventory. More precisely, our implementation allows to partition the cluster capacity into multiple inventories with customized sharing policies. This also allows us to run *Rayon* side-by-side queues managed by traditional scheduling policies.

For practical reasons the planner in our prototype assumes that resources are fungible, and thus: 1) ignores locality preferences, and 2) ignores granularity of requests. This means we treat resources as a non-differentiate continuum, and only check total availability of each resource type in the cluster. We rely on the scheduler to compensate for this at execution time. The use cases we investigate in § 3.4 are all well supported by this. These limitations can of course be lifted, but we defend their practical validity to bound the inventory complexity, and to avoid micro-management of allocations (likely moot due to dynamically fast evolving conditions).

In § 3 we showed that our placement problem is NP-complete. We thus focus on robust heuristics to search for a valid placement of jobs. 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.

To this purpose we leverage two observations: 1) atomic expressions connected by *order* operators with a single *window* operator, cover the most important classes of jobs, and 2) regardless of the allocation, the system can leverage spare capacity to anticipate work.

We design simple heuristics that explore the inventory with a single scan backward from the job's deadline, and allocates, respecting gang-needs, each stage of RDL expression in reverse order. This finds the "latest" place where the allocation is possible. At runtime, if resources are abundant we will try to anticipate as much work as possible, in a best-effort fashion. Committing late, and running early help us increase the chances, that jobs with tighter deadlines will be accepted. This covers all individual malleable and gang jobs with deadlines, and pipelines described in terms of their overall skyline.

As an example consider Figure 7, where we show the reservation and actual utilization[5] for a pipeline with three stages, corresponding to the following 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 and above its reservation. At time 650 the job completed and the ApplicationMaster can cancel the rest of its reservation, thus freeing up the inventory.

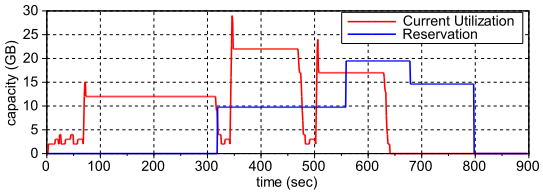

Figure 7: Allocation of a pipeline.

Simply reverting the direction in which we perform allocations (i.e., moving forward in time) we support another important use case, latency-sensitive allocations for gang-jobs (e.g., a Giraph job submitted interactively).

[5]For the sake of presentation, we project the multi-dimensional notion of capacity according to its main-memory axis only.

Note that this greedy algorithm 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 per second, from the SWIM workload [8, 9]. This handles the load of even the busiest clusters we studied.

Allocator. The allocator of Section 3.3.2 is implemented in YARN by a combination of a Plan Follower and a modified version of the CapacityScheduler. The PlanFollower implements Algorithm 1, and constantly reads the desired allocation from the inventory and propagates this to the CapacityScheduler. The CapacityScheduler has been modified in order to enable dynamic creation/reconfiguration/destruction of queues (normally a limited and heavy-handed process). In essence, our implementation tries to provide predictable resource allocation to jobs by dynamically reconfiguring the capacity scheduler queues.

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 a scalability concern even for very large and busy clusters.

Adapter. The API we expose to users allows for renegotiation, where "updating" a session is akin to an atomic delete and insert operation. The other concern for the adapter is to support replanning. Our current implementation provides a general purpose mechanism, triggered by the PlanFollower, whenever the cluster resources drift pass a certain tolerance window, and a simple policy. The policy leverages a greedy strategy that scans the inventory, and whenever a violation is detected cancel reservations active in that time-window untill the inventory constraints are respected. This is akin to off-line scheduling and exploring optimal policies in this context is an interesting future direction.

Quota management. We implemented the CapacityOverTimePolicy discussed in Section 3.3.4. This validation can be done in linear time by a single scan of the allocations of this user/group that overlap the time range from $first(a) - win$ to $last(a) + win$. To further speed-up execution we maintain a compact view of each user/group allocations, which allows us to cheaply compute the integral of resource utilization over a time-window. By sliding the time-window and adding/removing deltas we can cheaply verify that the user is not over-allocating.

5. EXPERIMENTAL EVALUATION

Rayon is designed with real-world needs and deployments in mind, so instead of delivering an extreme, though lob-sided, win on a single dimension, we strive to push for a balanced improvement across multiple important dimensions.

We deploy our prototype implementation of *Rayon* along with our modifications to YARN (version 2.1.0) on a 256-nodes cluster and run a two-part experimental evaluation.

The first part of the evaluation compares *Rayon* against the stock YARN CapacityScheduler (CS) using previously published cluster workloads from various companies [8, 9] (§ 5.2). We examine key cluster metrics and observe: 1) an almost tuning-knob free experience, 2) 0% SLA violations (vs 15% for the baseline) while accepting over 96% of SLA jobs, 3) latency is reduced for almost 40% of best-effort jobs, 4) 30 to 50% increase in cluster utilization, and 15% increase in cluster throughput.

The second part of the evaluation explores *Rayon*'s ability to support diverse workloads (§ 5.3), and the impact of over-reservation

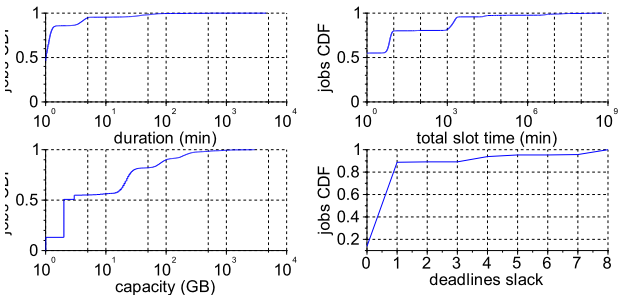

Figure 8: Job distribution for duration, total slot time, parallel capacity, and deadline slack.

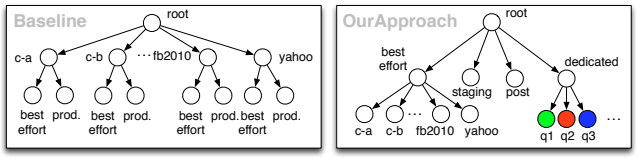

Figure 9: Visualization of the queue configuration.

(§ 5.4). For this purpose we leverage a rich mix of: Map-Reduce jobs, Giraph page-rank computations, and production pipelines with time-varying resource demands. From this set of experiments we learn that: 1) *Rayon* can achieve 100% cluster utilization despite job diversity, 2) 50% over-estimation in RDL requests is well tolerated, with a small 7% drop in job acceptance, near zero impact on cluster utilization, and 20% increase in best-effort jobs served (assuming sufficient pressure of best-effort jobs).

We differ to Appendix A for more experimental results on preemption, and simulations of different planning agents.

5.1 Experimental setup

Our experimental setup comprises of (1) cluster configuration and the software we deployed (Section 5.1.1) and (2) workloads (Section 5.1.2) used for the evaluation.

5.1.1 Cluster setup

Our large experimental cluster has approximately 256 machines grouped in 7 racks with up to 40 machines/rack. Each machine has 2 X 8-core Intel Xeon E5-2660 processors with hyper-threading enabled (32 virtual cores), 128GB RAM, 10Gbps network interface card, and 10 X 2-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 2.1.0 with our modifications for implementing *Rayon*. We use HDFS for storing job input/output with the default 3x replication. 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).

5.1.2 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).

Distribution-based Map-Reduce Workload The SWIM project [8, 9] provides detailed characteristics of Map-Reduce workloads from five Cloudera customers clusters, two Facebook clusters, and a Yahoo! cluster. The cluster sizes range from 100's of nodes up to

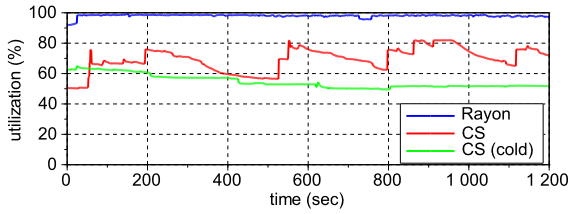

Figure 11: Cluster utilization, and preemption over time for: *Rayon*, the CapacityScheduler (CS), and the CapacityScheduler running with reduced load (CS cold).

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. We built a synthetic load generator which produce jobs from these distributions.

Synthetic Giraph Workload 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.

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 2). We represent each stage of the "skyline" using a synthetic MapReduce job.

Deriving SLAs Information about SLAs were not available from [8, 9], as today's system do not provide this feature. Based on conversation with cluster operators we settled for a conservative 5% of jobs with deadlines (see Figure 8), and a 10% "slack" (i.e., overestimation) over the actual job resource requirements, which were known since we control job ergonomics. For Microsoft production pipelines we talked with job owners and set SLAs accordingly.

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 Rayon versus YARN CapacityScheduler

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 [29]. In the experiment, we generated MapReduce jobs at the rate of 5,400 jobs per hour. Figure 9 shows the queue configurations for the CapacityScheduler, both for baseline and for *Rayon*. Each of the 8 workloads we run receives 2 queues (production and best-effort). The capacity associated with each queue is proportional to the exact total slot utilization that will be requested during the experiment—ideal condition. We follow best practices in configuring maximum capacity for each queue (up to 2x the guaranteed capacity). For *Rayon* we use a much simpler configuration (hence the tuning-free comment), where 70% of the cluster capacity is dedicated to production jobs, and we let *Rayon*'s dynamic allocation to subpartition it among production jobs. The remaining capacity is proportionally allocated to best-effort jobs like for the baseline. *Rayon* redistributes all unused resource among production and best-effort jobs (key to high utilization), and leverages preemption [29] to rebalance allocations when neeeded.

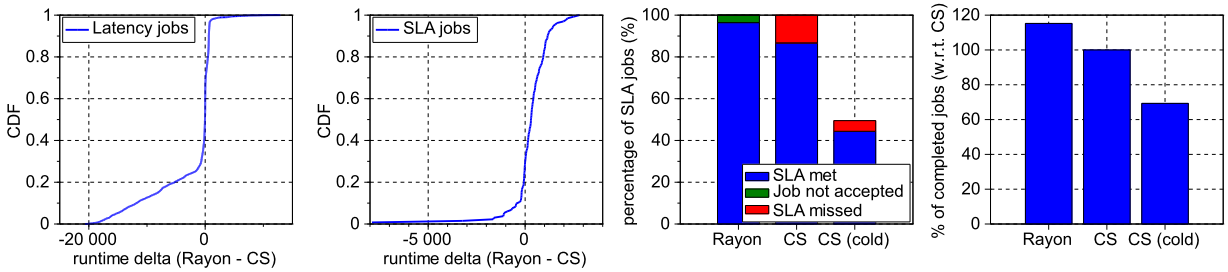

Figure 10: End-to-end experiment showing: 1) CDF of job runtime changes for Latency jobs, 2) CDF of job runtime for SLA jobs, 3) bar graph for jobs meeting or missing SLAs, and 4) total number of jobs completed during the experiment.

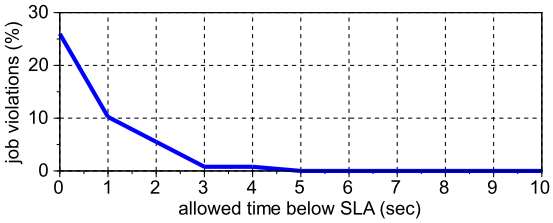

Figure 12: Percentage of job violations vs time allowed below nominal SLA.

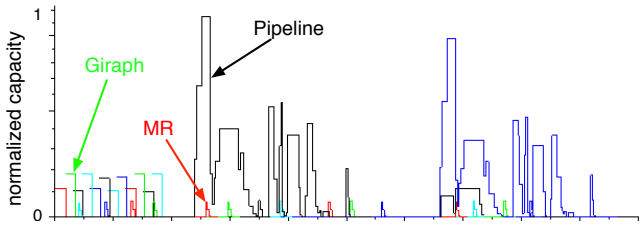

Figure 13: Visualization of dynamic queue allocations over time: including pipelines and gangs.

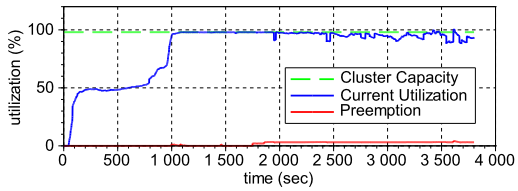

Figure 14: Cluster utilization when running a mixture of gang/fluid, besteffort/SLA jobs, and piepelines.

Figure 10 shows the results of our experiments when running a mix of production and best-effort jobs. *Rayon meets the resource allocation SLAs for all jobs it admits*, i.e., it delivers all resources requested by the job within the promise made by the reservation. Furthermore, by employing admission control, *Rayon* is able to reject jobs for which are going to provably miss their SLAs—less than 4% in our experiment. This immediate feedback, albeit negative, is invaluable to our users. In contrast, the stock YARN CapacityScheduler accepts all jobs but fails to meet the SLAs for roughly 15% of them. This means that with the stock YARN CapacityScheduler, the cluster has to be under-utilized to increase the odds that SLAs to production jobs are met. Figure 10 shows that even a massive 50% decrease in job submission rate (down to 2700 jobs/hour), does not completely fix missed SLAs—this is labelled "CS(cold)" in figure.

Figure 11 shows the cluster utilization achieved by the two systems. The stock YARN CapacityScheduler does not leverage preemption, and thus not all resources can be allocated when queues are temporarily out of balance. This yield lower utilization and it is consistent with several production environments we are aware of. On the other hand, since *Rayon* incorporates work-conserving preemption, we are able to maximize cluster utilization, despite temporary queue imbalances. 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.

The last key metrics we analyze, is how quickly can *Rayon* provide resources to a job that has a reservation. 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 throughout its entire runtime. Since jobs runtimes for production jobs far exceed 5 seconds we consider this a significant win.

5.3 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 [6]. 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.

Figure 14 shows the overall cluster utilization, during this experiment. After the initial warm-up phase utilization floats around 100%, as desired. The figures also shows small amount of preemption being used.

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

5.4 Impact of Over-reservation

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

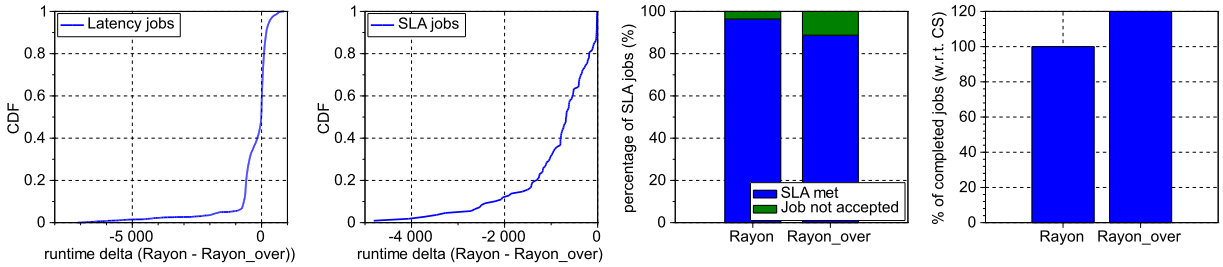

Figure 15: Effect of 50% over reservation (i.e., production jobs ask for 50% more resource reservations than they will use).

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 until completion (thus, subject to uncertainty).

Over-reservation, however, affects job acceptance. In Figure 15, we show what happens to our key metrics if we run the same workload as in Section 5.2, 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 re-distribution of resources allows it to cope with misconfigured reservations and maintain high cluster throughput while still meeting SLAs.

6. RELATED WORK

While many of the ideas we leverage in our system have been previously investigated in related areas, the novelty of our approach is in tackling the complexity of the overall problem, and to build and open-source a practical infrastructure. We are not aware of any system capable of handling the combination of classic batch jobs (flexible, scalable, fault-tolerant), jobs with gang requirements, complex pipelines with inter-stage dependencies, and yield high cluster utilization in consolidated environments. We organize the related work in four categories as follows.

Big-data resource management. Resource management has been an area of active investigation, with recent systems such as YARN [29], Corona [2], Omega [26] and Mesos [18] leading the charge, lifting the programming models limitations of previous systems, and proposing more extensible scheduling infrastructures. We argue that the framework we proposed could be adapted to each of these systems. For example, the data structures that Omega and Mesos use to represent the cluster state can be enhanced to explicitly handle time as a new dimension, and our planning agents could be embedded in the framework schedulers they support. Some other features, such as enforcement of sharing policies are more easily achieved in a centralized scheduler such as YARN [29].

Delay Scheduling [32], DRF [16], H-DRF [6], and Quincy [19] are recent examples of techniques for sharing resources among cluster tenants. Each of these influences or defines a correct allocation given fairness invariants, and provides support for locality awareness. All of these focus on instantaneous invariants, and do not support gang and skyline requirements. *Rayon* represents an ideal substrate to investigate temporal extension of these approaches—much like we did for the notion of capacity in Section 3.3.4.

All the big-data infrastructure we are aware of focus on scalability, fault-tolerance and flexible jobs, and provides no support for admission control, time-based SLAs, and gang requirements.

HPC resource management. HPC schedulers [28, 3, 27] cover a complimentary side of the spectrum, by focusing less on scalability and fault-tolerance (this makes them inadequate to support big-data workloads [29]), and more on gang semantics, and real-time scheduling of full-machines. Maui [3] prioritize jobs based on their desired start-times, and backfills the cluster with small jobs, like we do. Condor's notions of ClassAd, and the DAGMan scheduler, provide some of the features we are after, such as specifying resource requirements and a ranking function, and support for job dependencies. [24] provides clever workarounds to schedule MapReduce jobs on HPC clusters, but does not address the scalability issues reported in [29]. [7] is an interesting perl/mysql prototype with aims comparable to *Rayon*, but with no support for inter-job dependencies, nor big-data scalability.

Much of the above is possible in HPC settings, thanks to abundant use of preemption. Our recent work on providing work-preserving preemption for big-data clusters [4, 29] and parallel efforts in [10], provided us with enough scheduling flexibility to deliver SLAs and high-cluster utilization.

Deadlines and predictability. Prior work on SLOs for batch jobs has focused on profiling specific application frameworks (e.g., MapReduce [30], Scope [14]), and build ad-hoc schedulers. To the best of our knowledge, none of these systems provides a declarative language such as RDL, nor support a mixture of best-effort and deadline jobs, with and without gang and dependencies semantics. Of particular interest is [30], that provides carefully built profiles and models of MapReduce execution, and influence scheduling decisions to achieve completion deadlines. The key limitations is lack of support for gang, inter-stage dependencies, and more generally non-MapReduce jobs.

Bazaar [21] 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. Extending *Rayon*'s resource model to include network reservations is focus of ongoing work. Lucier et al. [22] 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.

The effort in [13] focus on supporting a mix of non-realtime and realtime workloads on big-data clusters. The authors present solutions for both resource definition and scheduling for the narrow class of MapReduce jobs. Our approach takes on a broader and

harder class of jobs, including gang and inter-job dependencies.

Similar techniques have also been explored in the context of cluster provisioning for IaaS [17].

Resource definition languages. The very idea of a declarative language for resource definition is not new, starting with IBM JCL and more recently SLURM [31]. In particular SLURM supports a rich set of algorithms for inferring job priority and mechanisms to evict, suspend, and checkpoint jobs based on those priorities. Comparatively *Rayon*'s reservations are very declarative and logical in nature, they expose flexibility that can be leverage dynamically by our system, while SLURM's reservation are relatively fixed, and capture exclusive and explicit access to nodes.

7. 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 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.

8. 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] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True Elasticity in Multi-Tenant Clusters through Amoeba. In *SoCC*, October 2012.

[5] 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*.

[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 *SoCC*, 2013.

[7] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard. A batch scheduler with high level components. In *CCGrid*, 2005.

[8] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads. *PVLDB*, 2012.

[9] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *MASCOTS*, 2011.

[10] 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.

[11] B.-G. Chun. Deconstructing production mapreduce workloads. In *Seminar at: http://bit.ly/1fZOPgT*, 2012.

[12] A. Desai, K. Rajan, and K. Vaswani. Critical path based performance models for distributed queries. In *Microsoft Tech-Report: MSR-TR-2012-121*, 2012.

[13] X. Dong, Y. Wang, and H. Liao. Scheduling mixed real-time and non-real-time applications in mapreduce environment. In *ICPADS*, 2011.

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

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

[16] 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.

[17] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *SOCC*, 2011.

[18] B. Hindman and et al. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[19] 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.

[20] 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.

[21] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *SoCC*, 2012.

[22] B. Lucier, I. Menache, J. S. Naor, and J. Yaniv. Efficient online scheduling for deadline-sensitive jobs: extended abstract. In *SPAA*, 2013.

[23] S. Narayanamurthy, M. Weimer, D. Mahajan, T. Condie, S. Sellamanickam, and S. S. Keerthi. Towards resource-elastic machine learning. In *BigLearn*, 2013.

[24] M. Neves, T. Ferreto, and C. Rose. Scheduling mapreduce jobs in hpc clusters. In *Euro-Par*. 2012.

[25] 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.

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

[27] G. Staples. Torque resource manager. In *IEEE SC*, 2006.

[28] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor: a distributed job scheduler. In *Beowulf cluster computing with Linux*, 2001.

[29] V. Vavilapalli and et al. Apache hadoop yarn: Yet another resource negotiator. In *SoCC*, 2013.

[30] A. Verma, L. Cherkasova, and R. H. Campbell. Aria: Automatic resource inference and allocation for mapreduce

environments. In *ICAC '11*, 2011.

[31] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, 2003.

[32] 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.

APPENDIX

A. EXPERIMENTS ADDENDUM

A.1 Work Preserving preemption

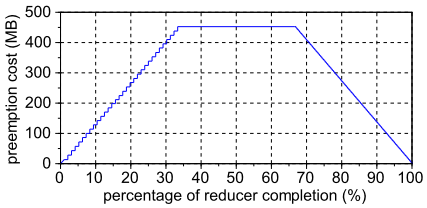

Figure 16: Preemption cost as a function of reducer progress.

In this appendix we provide 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 [29]. This is also comparable with prior work in [4] and parallel efforts in [10]. 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 16, 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.

A.2 Admission Control: Simulations

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

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 17, 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 17 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

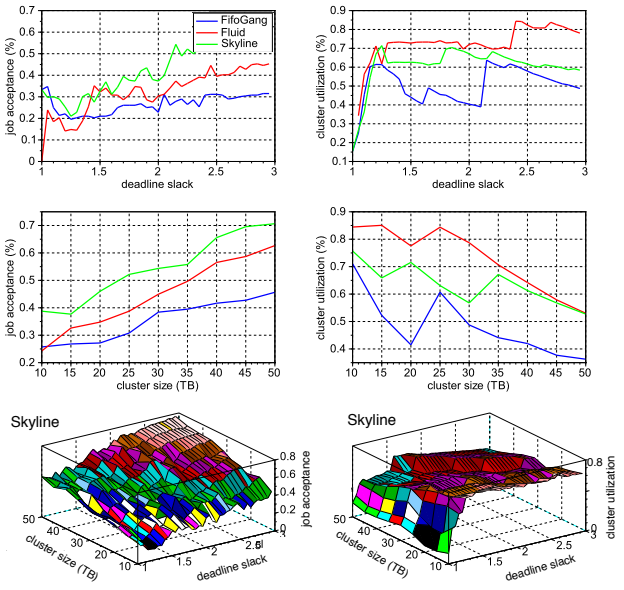

Figure 17: Simulating the planning of 7200 jobs/hour.

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 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.

B. PLANNING

In this Appendix, we complement the discussion of planning by presenting a general algorithm for placement of RDL expressions, and with an ILP formulation for our placement problem. This material is meant as a general addendum to hint at the generality of the problem at hand, and should act as an inspiration for researchers to explore more advanced solutions for the problems we tackle heuristically in Rayon.

B.1 A general algorithm for RDL placement

Algorithm 2, shows a general purpose algorithm to satisfy an RDL expression.

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.

The algorithm receives in input the RDL expression r, and an inventory I of cluster resources, and returns a valid placement for the request r in I if one exists, or rejects the job all together. The inventory is a representation of cluster resources and the current placement of previously accepted jobs for every moment in time from now up to a given time-horizon.

Algorithm 2: 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)` *:* $(Inv, Inv) \rightarrow Inv$`)` **:** Inv
  $inv' \leftarrow$ **nil**
  forall $s \in$ `accept(`*e, inv*`)`
    $inv' \leftarrow$ `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(`$\{e_1, \ldots, e_2\}$ *: any, inv : Inv*`)` **:** Inv
  forall $s \in$ `accept(`e_1, inv`)`
    yield s
  `accept(`$any\{e_2, \ldots, e_n\}, inv$`)`
def `accept(`$\{e_1, \ldots, e_2\}$ *: all, inv : Inv*`)` **:** Inv
  forall $s \in$ `accept(`e_1, inv`)`
    yield `accept(`$all\{e_2, \ldots, e_n\}, s$`)`
  nil
def `accept(`$\{e_1, \ldots, e_2\}$ *: order, inv : Inv*`)` **:** Inv
  forall $s \in$ `accept(`e_1, inv`)`
    yield `accept(`$order\{e_2, \ldots, e_n\}$, *inv[*`last(`$e_1, s$`)`, $inv.f$`)`
  nil
def `accept(`$\langle l, h, w \rangle$ *: atomic, inv : Inv*`)` **:** Inv
  `//` **yield** `valid placements in` inv
  nil
def `last(`*e : RDL, inv : Inv*`)` **:** Int
  `//` `last allocation to` e `in` inv

Algorithm 2 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.

B.2 Integer Linear Programming model of our placement problem

In this section, we provide a Mixed-Integer Linear Programming formulation for the planning component of our problem. The formulation is slightly richer that what we discussed so far and captures constraints for: 1) job with soft and hard deadlines, 2) dependencies among job allocations, 3) time and capacity gang semantics, 4) planned node addition/decommissioning, and optimize an objective function which juggle user and system preferences in placement.

In order to be comprehensive the formulation leverages expensive optimization constructs (integrality and large number of variables and constraints). This formulation is therefore used as a conceptual blueprint to build a practical system. In our YARN-based implementation of this design we rely on robust, fast heuristics and aggressive replanning as resilient stunt doubles for this ILP formulation.

The formulation is given as follows:

$$\text{minimize } \alpha \sum_{i>d_j} c_{ij} * x_{ij} + \beta \sum_{i<d_j} u_{ij} * x_{ij} + \gamma \sum_{ij} abs(\frac{dx_{ij}}{di})$$

subject to:

$$\forall i : \sum_j x_{ij} \leq R_i \tag{1}$$

$$\forall j \in H : \sum_{s_j \leq i < d_j} x_{ij} = w_j \tag{2}$$

$$\forall j \notin H : \sum_{i \geq s_j} x_{ij} = w_j \tag{3}$$

$$\forall i \forall j : \; x_{ij} < par_j \tag{4}$$

$$\forall i \forall j \in F : (x_{ij} \in \mathbb{R};\; x_{ij} \geq 0; x_{ij} \leq \bar{R}) \tag{5}$$

// inter job dependencies

$$\forall i \forall j : s_{ij} = s_{i-1j} + x_{ij} \tag{6}$$

$$\forall i \forall j : f_{ij} = f_{i+1j} + x_{ij} \tag{7}$$

$$\forall i \forall (j', j'') \in D : f_{ij'} + s_{ij''} < g_{j'} + g_{j''} \tag{8}$$

// gang semantics

$$\forall i \forall j \in G : x_{ij}, s_{ij}, f_{ij} \in \{0, g_j\} \tag{9}$$

$$\forall i \forall (j', j'') \in D_g : f_{ij'} + s_{ij''} > 0 \tag{10}$$

Where the goal of the solver is to determine the optimal value of the variables x_{ij} that capture the portion of cluster capacity allocated for job j at every time interval i. Where j ranges on all jobs in the system, and i ranges from zero (now) to the time horizon of the planner, while not assumptions are made in this formulation, one can envision i to have 1-minute granularity steps for 1 week.

Constraints. We introduce the rest of the notation while commenting of the constraints.

1 guarantees the cluster is never overcommitted, by enforcing the sum of assignment is below the cluster capacity R_i at any time i, proper values of R_i can be used to capture planned addition/decommissioning of nodes.

2 for all jobs with hard deadlines $j \in H$ we guarantee that the sum of the allocations between the job earliest valid start-time s_j and deadline d_j is equal to the job overall amount of work w_j.

3 similarly for job with soft deadlines we impose allocations are after s_j, while we rely on the optimization function to reduce the amount of work done after the deadline.

4 guarantees that allocations never violate the maximum parallelism par_j that a job is capable of.

5 for all jobs with "fluid" requirements, i.e., not requiring gang, we allow x_{ij} to range on real values between zero and the maximum value $\bar{R}$ of R_i.

Next we describe the constraints used to selectively capture gang and dependecy semantics for jobs that require them, i.e., $j \in G$. The key intuition is to treat each "quanta" of work as an independent job and impose that the values of x_{ij} only assume zero or g_j values (where g_j is the gang requirement for that job, and

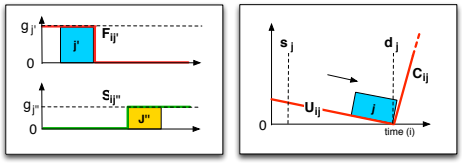

Figure 18: Visual representation of start and finish functions: s_{ij} and f_{ij}, used in the ILP formulation. They are used to enforce job dependencies, e.g., J1 is before J2, is captured as $f_{i1}) + s_{i2} \leq \bar{R}$.

impose ordering and (optionally) lack of gap between their allocations. This allows to build arbitrary skylines, and job dependencies.

6-7 construct two support variables s_{ij} and f_{ij}, which we visualize in Figure 18 capturing the start time and end time of allocations for a job.

8 imposes a "happens before" constraint between pairs of jobs $(j', j'') \in D$, where D is the set of pairwise job dependencies. This is done by imposing a constraints on the sum of end function for j' and start function for job j'', which can only be satisfied if j' is scheduled before j''.

9 bounds all functions involved in dependencies and gang to assume integral values that match the job gang requirement, i.e., zero or g_j.

10 this is an optional constraints used to enforce time-based gang requirements, i.e., that a job receives an interrupted set of time slots. This is done by imposing no gaps between the allocations of all quantas $(j', j'') \in D_g$ where $D_g \subseteq D$ is the set of dependecies capturing time-based gang requirements. The no-gap is represented as a non-zero requirement over the sum of end and start functions for j' and j''.

Note that, constraints 6-8 work correctly in combination with the 9, i.e., when we consider jobs as a set of discrete quanta of unit-time duration and bound to assume values 0 or g_j. The formulation can be relaxed to use $\geq$ constraints for 6,7 and adding a component in the objective function that minimizes the s_ij and f_ij. This would allow non-discretized management of x_ij in exchange for a more complex objective function.

Objective Function. In the objective function we need to balance three objectives: 1) minimize soft deadlines violations, 2) pack jobs according to some time preference, 3) minimize the need for preemption. This is achieve by balancing the three components of the objective function, by tuning the factors α, β, γ. Since we use this formulation only as a conceptual model we omit details on the tuning of such parameters.

The first component $\alpha \sum_{i>d_j} c_{ij} * x_{ij}$ captures the cost c_{ij} (which we are trying to minimize) of scheduling jobs past their deadlines d_j. The weight c_{ij} are shown in Figure 18b. The further away from the deadline the more expensive is to schedule a job. This steep curve pushes the solver to look for solutions that respect soft deadlines as much as possible.

The second component $\beta \sum_{i<d_j} u_{ij} * x_{ij}$ captures our preference to schedule jobs earlier or later. Here our formulation differs from the common sense "schedule jobs as early as possible", and we push jobs as close to their deadline as possible. This is motivated by: 1) need to run besteffort, latency-critical jobs on the same cluster, and 2) by the fact that we are only affecting guaranteed resource allocations, while at runtime work will be anticipated as much as possible. Different choices for this portion of the objective function can affect scheduling based on other preferences.

The third and final component $\gamma \sum_{ij} abs(\frac{dx_{ij}}{di})$ we include in our minimization function, aims at limiting the amount of preemption, by favoring solutions with low churn. This is captured formally by minimizing the absolute value of the derivative of our allocation variables x_ij. For the sake of presentation we use derivative and absolute value notations in the objective functions, however this can be linearized by: 1) introducing a new set of variables y_{ij} and add constraints bounding them to be the derivative of x_ij, e.g., $y_{(1,2)j} = x_{2j} - x_{1j}$. Absolute value can be linearized by introducing a variable that is greater than both direction of the sum, e.g., a−b is captured by a variable $k \geq a - b$; and $k \geq$ b−a, by minimizing k only one of the two constraints will be active at any given time (the positive one, and k will be forced to take that value, which is also equal to a−b.

Note that the one presented is an example of multi-goal objective function, that balances user preferences (run early/late) and system preferences (lower preemption costs). The modular structure of Rayon would allow this or many other placement preferences to be tested on top of a working big-data system.