TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters

Alexey Tumanov* Timothy Zhu* Jun Woo Park*
Michael A. Kozuch† Mor Harchol-Balter* Gregory R. Ganger*
Carnegie Mellon University*, Intel Labs†

Abstract

TetriSched is a scheduler that works in tandem with a calendaring reservation system to continuously re-evaluate the immediate-term scheduling plan for all pending jobs (including those with reservations and best-effort jobs) on each scheduling cycle. *TetriSched* leverages information supplied by the reservation system about jobs' deadlines and estimated runtimes to *plan ahead* in deciding whether to wait for a busy preferred resource type (e.g., machine with a GPU) or fall back to less preferred placement options. Plan-ahead affords significant flexibility in handling mis-estimates in job runtimes specified at reservation time. Integrated with the main reservation system in Hadoop YARN, *TetriSched* is experimentally shown to achieve significantly higher SLO attainment and cluster utilization than the best-configured YARN reservation and CapacityScheduler stack deployed on a real 256 node cluster.

1. Introduction

Large clusters serving a mix of business-critical analytics, long-running services, and ad hoc jobs (e.g., exploratory analytics and development/test) have become a data center staple. Many of the analytics jobs have strict time-based SLOs [7], and services have time-varying but high-priority resource demands, while ad hoc jobs are generally treated as best effort with a desire for minimal latency for the waiting user. Satisfying all of these effectively requires that a cluster scheduler exploit knowledge about job characteristics to create effective plans for resource allocation over time.

Fortunately, the business critical activities that dominate cluster usage also tend to come with a degree of predictability. For example, studies consistently confirm that production jobs consume over 90% of cluster resources [4–6, 35], and most of them are workflows submitted periodically by automated systems [19, 32] to process data feeds, refresh models, and publish insights. They are often large and long-running, consuming tens of TBs of data and running for hours, and they come with strict completion deadlines [7]. Because they are run regularly, research confirms that the production tools developed to support them can robustly predict job runtimes as a function of resource types and quantities [1, 7, 10–12, 38].

A scheduler armed with such information should be able to make much better decisions regarding current and planned future resource allocation. Unfortunately, current schedulers fall far short of ideal, failing to include interfaces for accepting much of the available information and lacking algorithms for exploiting it. Taking the Hadoop/YARN cluster scheduling framework (used in many production data centers) as a concrete example, its runtime scheduler is coupled with a reservation system that supports time-based resource reservations informed by job runtime estimates. But, the runtime scheduler fails to use that information, enforcing a static reservation plan instead. As a result, it is unable to leverage any resource type or start-time flexibility that may exist. What is needed is a scheduler that understands per-job requirements, tradeoffs, and runtime estimates and exploits that information to simultaneously optimize near-term decisions and the longer-term plan in the face of unknown future job submissions.

This paper introduces TetriSched, a cluster scheduler that (a) leverages runtime estimate and deadline information supplied by a deployed reservation system, (b) supports time-aware allocation of heterogeneous resources to a mixture of SLO + best effort workloads with placement preferences ranging from none to combinatorially complex, (c) simultaneously evaluates the needs of all pending jobs to make good global allocation decisions, rather than greedy job-by-job decisions, and (d) re-evaluates its resource space-time plan each scheduling cycle in order to adapt to job arrivals, changing cluster conditions, and imperfect job runtime estimates.

TetriSched is integrated into YARN such that it receives best effort jobs directly and SLO jobs via the user-facing reservation system built into YARN. TetriSched introduces

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

ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2901318.2901355

an expressive language, which we call Space-Time Request Language (STRL), for declarative specification of preferences in resource space-time; the STRL expressions can be automatically generated for SLO jobs relying on the reservation system. Each scheduling cycle, TetriSched aggregates pending resource requests and automatically translates them to a Mixed Integer Linear Programming (MILP) formulation, allowing an MILP solver to consider the set as a whole. The solver runtime can be bounded, with minor loss of optimality, but we find that it is not necessary even for our 256-node cluster experiments. The MILP formulation explicitly considers the job runtime estimates, using them to plan ahead regarding expected future resource availability. Importantly, this plan-ahead is repeated each scheduling cycle, to avoid being restricted to previous plans when conditions change (e.g., new jobs arrive or a job runtime estimate is wrong).

Experiments confirm TetriSched's effectiveness. For example, on a workload derived from production workload traces, TetriSched outperforms the capacity scheduler currently used with Rayon in Hadoop/YARN on each of the success metrics. It satisfies up to 3.5× more SLOs while simultaneously lowering average best effort job latency by as much as 80%. TetriSched provides the largest benefits in the more difficult situations, such as when dealing with imperfect runtime estimates, complex forms of resource heterogeneity, and high load. Regular re-evaluation of the scheduler plan, combined with simultaneous consideration of pending jobs, allows TetriSched to exploit the knowledge of flexibility in resource and time needs to more fully exploit available resources at each point.

This paper makes the following primary contributions: (1) It introduces Space-time request language (STRL), an expressive language for declarative specification of preferences in resource space-time. (2) It describes how integration with a reservation system for production jobs with SLOs provides key information needed to construct STRL expressions. (3) It describes an algorithm for translating a set of resource requests to a Mixed Integer Linear Programming (MILP) formulation that can be efficiently solved. (4) It describes how using plan-ahead together with regular re-planning leverages runtime estimates to consider future options without hard-coding a potentially wasteful plan given dynamic conditions. (5) It presents an evaluation of the YARN-integrated TetriSched implementation on a sizable (up to 256 nodes) real cluster with a mixture of gridmix-generated workloads based on traces and other information about production jobs from Facebook, Yahoo, and Microsoft. The evaluation demonstrates that TetriSched is more effective than YARN's default scheduler and, especially, that TetriSched is much better able to cope with imperfect information about jobs, deriving significant benefit without overly relying on it.

2. Background and Desired Features

We distinguish between three system features in the scheduling literature: (a) capacity reservation, (b) placement, and (c) ordering. We define ***reservation*** as the ability of a system to reserve and guarantee future resource capacity. Reservation systems promise future access to resources and serve as admission control frontends. They can also be used for capacity planning. Reservations should generally be regarded as longer-term resource planning. ***Placement*** is an act of assigning a set of specific resources to a job. A job is *placed* when it is mapped to a node or a set of nodes, where the local OS will orchestrate the execution of the job's individual tasks it's comprised of. Placement concerns itself with cluster resource `space`—their types, sets, quantities, and topology. ***Ordering*** is an act of determining which job is considered for placement next. A job is *ordered* when it is selected for placement next. Ordering concerns itself with cluster resource `time`—how much time is allocated to each job and in what sequence in time resource requests are satisfied. Scheduling, then, is the combination of placement and ordering and should generally be regarded as shorter-term resource allocation, ultimately deciding *who runs where next*.

2.1 Reservation vs. scheduling

While reservation systems and scheduling systems sound similar in nature, they actually solve different, complementary problems. Reservation systems such as Rayon [7]—the state-of-the-art YARN reservation system—are designed to guarantee resource availability in the long term future. It serves as an admission control system to ensure that resource guarantees are not overcommitted. By contrast, a scheduling system, such as TetriSched, is designed to make short-term job placement and ordering decisions. Scheduling systems are responsible for optimizing job placement to more efficiently utilize resources. Schedulers are also responsible for deciding when to run jobs without reservations, while ensuring that jobs with reservations are completed on time. Thus, scheduling systems, such as TetriSched, address a different, complementary problem than reservation systems, such as Rayon, and we run TetriSched in tandem with Rayon.

2.2 Placement considerations

To accommodate the specific needs of diverse distributed applications [9, 27, 28, 30], cluster resources have become increasingly heterogeneous, which makes job placement a much more challenging task. Different types of heterogeneity exist and should be considered in modern datacenters: static, dynamic, and combinatorial.

Static heterogeneity refers to diversity rooted in the static attributes of cluster resources, such as different processors, different accelerators (e.g., a GPU), faster disks (SSD), particular software stacks, or special kernel versions [30]. Static heterogeneity is on the rise and is only expected to increase [17, 20]. *Dynamic* or runtime heterogeneity refers to differences between cluster resources induced by the workloads themselves. For example, data-intensive computation frameworks, such as Hadoop MapReduce and Spark, derive performance benefits from data-locality. From such jobs' perspective, a set of nodes becomes heterogeneous when viewed

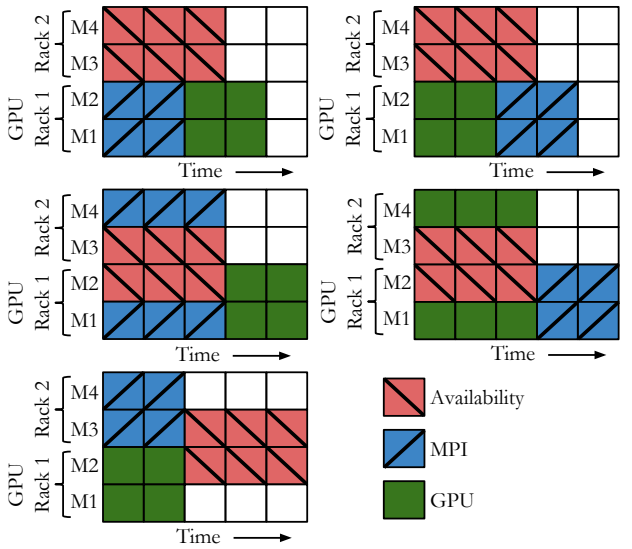


Figure 1. Five potential schedules for 3 jobs. Each grid shows one potential space-time schedule, with machines along the rows and time units along the columns. Each job requests 2 servers, and its allocation is shown by filling in the corresponding grid entries. The cluster consists of 2 racks each with 2 servers, and rack 1 is GPU-enabled. The Availability job prefers 1 server per rack. The MPI job runs faster if both servers are in one rack (2 time units) than if they are not (3 time units). The GPU job runs faster if both servers have GPUs (2 time units) than if they don't (3 time units).

through the lens of data locality. Machines with desired data become preferred to machines without it. Intereference [9, 10] is another form of dynamic heterogeneity that affects job performance when ignored. Static and dynamic heterogeneity intuitively creates a differentiated view of cluster resources from jobs' perspective. A given node (or a set of nodes) could be *unsuitable*, *acceptable*, or *desirable*, depending on how it fulfills the job's needs. Expressing these placement options and associated tradeoffs (known as "soft" constraints) is both challenging and necessary, as consolidated frameworks compete for the ever-changing landscape of increasingly heterogeneous cluster resources. Fig. 1 illustrates the complexity of supporting soft placement constraints even at the toy cluster scale.

Furthermore, as exemplified by HPC and Machine Learning applications with tightly-coupled communicating tasks, different *subsets* of machines may influence job execution differently. Some subsets (e.g., rack-local collections of k machines [33], data-local samples of data servers [36]) may speed up the execution or improve application QoS, while others may degrade it. We refer to such heterogeneity as *combinatorial*. Jobs that prefer all k tasks to be simultaneously co-located (e.g. MPI job in Fig. 1) in the same locality domain of the many available exemplify combinatorial constraints. Their preference distribution is over a superset of cluster nodes, in contrast to server-types or server quantities alone, which is a challenge to express and support. TetriSched captures all such placement considerations succinctly with STRL (Sec. 4.1).

2.3 Temporal considerations

2.3.1 Temporal constraints and runtime estimates

In addition to a description of the possible placement options, job scheduling requests are often associated with two other pieces of information: Service Level Objectives (SLOs) and estimated runtimes. SLOs specify user-desired constraints on job execution timing. Completion-time oriented jobs, for example, may specify that a job must complete by 5:00PM, or within an hour. For TetriSched to schedule meaningfully with such constraints, jobs must also have associated runtime estimates. Prior work [1, 7, 10–12, 38] has demonstrated the feasibility of generating such estimates; this paper will demonstrate that TetriSched is resilient to mis-estimation.

2.3.2 Planning ahead

TetriSched is able to *plan ahead* by leveraging information about estimated runtimes and deadlines provided by a reservation system. Knowledge of expected completion times for currently running jobs provides visibility into preferred resource availability. This makes it possible to make informed deferral decisions on behalf of pending jobs, if the overall queueing delay thus incurred is offset by the benefit of better performance on preferred resources. Without plan-ahead, the only options available are (a) to accept a sub-optimal allocation or (b) to wait indefinitely for desired placements, hoarding partial allocations, which wastes resources and can cause deadlocks. With plan-ahead, TetriSched obviates the need for jobs to hoard resources to achieve their optimal placement.

2.3.3 Planning ahead adaptively

It is important to emphasize that TetriSched's plan-ahead does not lock in future allocation decisions. Instead, they are re-evaluated on every scheduling cycle for all pending jobs. This is beneficial for a variety of reasons. First, new jobs can arrive at any time. More urgent deadlines may be accommodated if a job initially deferred to start at the next cycle is delayed further, but still meets its deadline. Not replanning can easily lead to a violation of that job's SLO as it's forced to wait longer. Second, when job runtimes are misestimated, the future schedule is improved by re-planning. If the runtimes of any jobs were underestimated, the start times of later jobs may be delayed as a consequence, running the risk of subsequent jobs missing their SLOs. If runtimes are overestimated, available capacity may be wasted. In either case, adaptive replanning is crucial.

2.4 Global scheduling

Many schedulers consider jobs one at a time in a greedy fashion. In heterogeneous clusters in particular, this can lead to sub-optimal decisions (see Sec. 7.2, Fig. 10) as the earlier scheduled jobs may, unnecessarily, take resources needed for

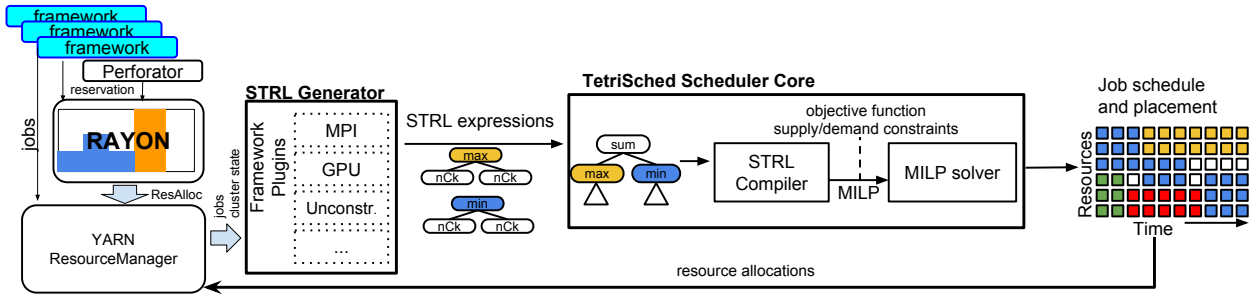


Figure 2. TetriSched system architecture

later-arriving jobs. To make better scheduling decisions, it is important to consider all pending jobs queued in the system. By simultaneously considering the placement and temporal preferences of all the jobs, TetriSched is able to make the best tradeoffs. We refer to this feature as global scheduling.

3. TetriSched

This section describes the architecture and key components of TetriSched. It works in tandem with the Rayon [7] admission control system, but continuously reevaluates its own space-time schedule separately from the reservation plan to adapt to system and job specification imperfections. TetriSched leverages information about expected runtimes and deadline SLOs supplied to Rayon via reservation requests, as well as its own heterogeneity-awareness and plan-ahead, to maximize SLO attainment, while efficiently using available cluster resources. Instead of the common greedy job placement in cluster schedulers today, TetriSched makes a global placement decision for all pending jobs simultaneously, translating their resource requests to an internal algebraic expression language that captures heterogeneity considerations and available time information for each job.

3.1 System Architecture

Fig. 2 shows the major components of our scheduling system stack and how they interact. Framework ApplicationMasters initially submit reservation requests for SLO jobs. Best-effort jobs can be submitted directly to YARN's ResourceManager without a reservation. We implement a proxy scheduler that plugs into YARN's ResourceManager in Fig. 2 and forwards job resource requests asynchronously to TetriSched. Its first entry-point is the STRL Generator that uses framework-specific plugins to produce STRL expressions used internally to encode space-time resource requests. To construct STRL expressions, The STRL Generator combines the framework-specified reservation information, such as runtime estimates, deadlines, and the priority signal (e.g., accepted vs. rejected reservations), with ApplicationMaster-specified job type to construct STRL expressions.

Resource requests are subsequently managed by TetriSched, which fires periodically. At each TetriSched cycle, all outstanding resource requests are aggregated into a single STRL expression and converted into an MILP formulation by the STRL Compiler. Solving it produces the job schedule that maps tasks for satisfied jobs to cluster nodes. This allocation decision is asynchronously communicated back to the ResourceManager (Sec. 3.3). YARN RM then takes over the job's lifecycle management. Specifically, as YARN NodeManagers heartbeat in, they are checked against the allocation map our proxy scheduler maintains and are assigned to ApplicationMasters to which they were allocated by TetriSched. Thus, we provide a clear separation of resource allocation policy from cluster and job state management, leaving the latter to YARN. We discuss the integration of TetriSched in the YARN framework in Sec. 3.3.

3.2 Scheduler

The scheduler's primary function is to produce space-time schedules for currently queued jobs. On each scheduling cycle, TetriSched aggregates STRL expressions with the SUM operator and assigns cluster resources so as to maximize the aggregated STRL expression value (see Sec. 4). Value functions are a general mechanism that can be used in a variety of ways [34], such as to apply job priorities, enforce budgets, and/or achieve fairness. The configurations in this paper use them to encode deadline sensitivity and priority differences (Sec. 6.2.2), producing the desired effects of (a) prioritizing SLO jobs, while (b) reducing completion times for best-effort jobs. The aggregated STRL SUM expression is automatically converted into an MILP problem (Sec. 5) by the STRL Compiler and solved online by an off-the-shelf IBM CPLEX solver at each cycle. Solver complexity is discussed and evaluated in Sec. 7.3.

3.2.1 Plan-ahead

One of TetriSched's novel features enabled by its support for space-time soft constraints is its ability to consider and choose deferred placements when it's beneficial to do so. We refer to this ability as "plan-ahead". Plan-ahead allows TetriSched a much wider range of scheduling options. This is particularly important for the scheduler to know whether it should wait for preferred resources (in contrast to never waiting [33] or always waiting [41]). Planning to defer placement too far into the future, however, is computationally expensive and may provide diminishing returns. Indeed, an increased plan-ahead interval leads to more job start time choices (see Fig. 1) and, correspondingly, larger MILP problem sizes. On each cycle, only the jobs scheduled to start at the current tick are extracted from the schedule and

launched. The remaining jobs are re-considered at the next cycle. Thus, placements deferred far into the future are less likely to hold as decisions are reevaluated on each scheduling cycle.

To support plan-ahead, no changes to the scheduling algorithm were needed. Instead, we leverage the expressivity of STRL expressions and construct them for all possible job start-times in the plan-ahead window. Since we quantize time, the resulting size of the algebraic expression is linear in the size of the plan-ahead window. A job's per-quantum replicas are aggregated into a single expression by the STRL Generator. The STRL Generator performs many possible optimizations, such as culling the expression growth when the job's estimated runtime is expected to exceed its deadline.

3.2.2 MILP Solver

The internal MILP model can be translated to any MILP backend. We use IBM CPLEX in our current prototype. Given the complexity and size of MILP models generated from STRL expressions (hundreds of thousands of decision variables on a 1000 node cluster with hundreds of jobs), the solver is configured to return "good enough" solutions within 10% of the optimal after a certain parametrizable period of time. Furthermore, as the plan-ahead window shifts forward in time with each cycle, we cache solver results to serve as a feasible initial solution for the next cycle's solver invocation. We find this optimization to be quite effective.

3.3 YARN Integration

For ease of experimentation and adoption, we integrate TetriSched into the widely popular, active, open source YARN [35] framework. We add a proxy Scheduler that interfaces with the TetriSched daemon via Apache Thrift RPCs. The interface is responsible for (a) adding jobs to the TetriSched pending queue, (b) communicating allocation decisions to YARN, and (c) signaling job completion to TetriSched. TetriSched makes allocation decisions based on thus provided information and its own view of cluster node availability it maintains.

4. Space-Time Request Language (STRL)

STRL's design is governed by the following five requirements that capture most practical workload placement preferences encountered in datacenters [30] and HPC clusters [26]: **[R1]** space-time constraints, **[R2]** soft constraints (preference awareness), **[R3]** combinatorial constraints, **[R4]** gang scheduling, and **[R5]** composability for global scheduling.

Intuitively, we need a language primitive that captures placement options in terms of the types of resources desired (encoded with equivalence sets defined in Sec. 4.2), their quantity, when, and for how long they will be used **[R1]**. This is captured by the STRL's principal language primitive called "n Choose k" (nCK). This primitive concisely describes resource space-time allocations. It eliminates the need to enumerate all the $\binom{n}{k}$ k-tuples of nodes deemed equivalent by the job. This primitive alone is also sufficient for expressing hard constraints. Soft constraints **[R2]**—enumerating multiple possible space-time placement options—are enabled by the MAX operator that combines multiple nCk-described options. MAX and MIN operators are also used to support simpler combinatorial constraints **[R3]** such as rack locality and anti-affinity. More complex combinatorial constraints can be achieved with SCALE and BARRIER, such as high availability service placement with specified tolerance threshold for correlated failures [34]. Examples include a request to place up to, but no more than, k' borgmaster servers in *any* given failure domain totaling k servers. The SUM operator enables global scheduling **[R5]**, batching multiple child expressions into a single STRL expression. The intuition for this language is to create composable expression trees, where leafs initiate the upward flow of value, while intermediate operator nodes modify that flow. They can multiplex it (MAX), enforce its uniformity (MIN), cap it, or scale it. Thus, a STRL expression is a function mapping *arbitrary* resource space-time shapes to scalar value. Positive value means the STRL expression is satisfied.

Limitations: STRL can express any constraint that can be reduced to resource sets and attributes. This includes inter-job dependencies (e.g. anti-affinity), if one of the jobs is already running. STRL cannot express inter-job dependencies for pending jobs, nor does it support end-to-end deadlines for a pipeline of jobs.

4.1 STRL Specification

This subsection formally defines STRL leaf primitives and non-leaf operators. A STRL expression can be:

1. an "n Choose k" expression of the form
nCk(equivalence_set, k, $start$, dur, v).
It is the main STRL primitive used to represent a choice of any k resources out of the specified equivalence set, with the start time and estimated duration. In Fig. 1, the GPU job's ask for $k=2$ GPU nodes would be $nCk(\{M1,M2\},k=2,start=0,dur=2,v=4)$, where v quantifies the value of such an allocation.

2. a MAX expression of the form $max(e_1,...,e_n)$. It is satisfied if at least one of its subexpressions returns a positive value. MAX carries the semantics of OR, as it chooses one of its subexpressions (of maximum value). MAX is used commonly to specify choices. In Fig. 1, the GPU job's choice between an exclusively GPU node allocation and any other 2-node allocation is captured as $max(e_1,e_2)$ where:
$e1 = nCk(\{M1,M2\},k=2,start=0,dur=2,v=4)$
$e2 = nCk(\{M1,M2,M3,M4\},k=2,start=0,dur=3,v=3)$
Iterating over possible start times in the pictured range($start \in [0,4]$) adds more placement options along the time dimension as well. General space-time elasticity of jobs can be expressed using MAX to select among possible 2D space-time shapes specified with nCk. Enumeration of options drawn from the same equivalence set with the same duration, but different k can be suppressed with the optional "Linear nCk" primitive (see [34] for complete formal specification).

3. a MIN expression of the form $min(e_1,...,e_n)$ is satisfied if all subexpressions are satisfied. MIN is particularly useful for specifying anti-affinity. In Fig. 1, the Availability job's primary preference (simultaneously running its 2 tasks on separate racks) is captured with the MIN expression as follows: $min(nCk(rack1,k=1,s=0,dur=3,v),nCk(rack2,k=1,s=0,dur=3,v))$. Here, each of the two subexpressions is satisfied iff one node is chosen from one of the nodes on the specified rack. The entire MIN expression is satisfied iff both nCk subexpressions are satisfied. This results in the allocation of exactly one task per rack.

4. a BARRIER expression of the form $barrier(e,v)$ is satisfied if the expression e is valued v or more. It returns v when satisfied.

5. a SCALE expression of the form $scale(e,s)$ is satisfied if subexpression e is satisfied. It is a unary convenience operator that serves to amplify the value of its child subexpression by scalar s.

6. a SUM expression of the form $sum(e_1,e_2...,e_n)$ returns the sum of the values of its subexpressions. It is satisfied if at least one of the subexpressions is satisfied. The sum operator is used to aggregate STRL expressions across all pending jobs into a single STRL expression.

4.2 Equivalence sets

An important notion in TetriSched is that of *equivalence sets*, which are equivalent sets of machines from the perspective of a given job. For example, a job that prefers to run on k nodes with a GPU may equally value any k-tuple of GPU nodes, while valuing lower elements outside of that set. The ability to represent sets of machines that are equivalent, from a job's perspective, greatly reduces the complexity of the scheduling problem, as it obviates the need to enumerate all combinatorial choices of machines. Equivalence sets are instrumental to the reduction of combinatorial complexity. Instead of enumerating all the possible spatial choices (e.g, any 5 nodes on a 40 node rack), we only have to specify the set to choose from and how much to choose.

4.3 STRL examples

Suppose a GPU job arrives to run on a 4-node cluster in Fig. 1. We have 4 nodes, with M1,M2 with a GPU and M3,M4—without. A GPU job takes 2 time units to complete on GPU nodes and 3 time units otherwise. The framework AM supplies a value function $v_G()$ that maps completion time to value. A default internal value function can be used, if not specified (as done in our experiments). For each start time s in $[S, Deadline]$—the interval extracted from the Rayon RDL expression, we have the following choices:

$nCk(\{M1,M2\},k=2,s,dur=2,v_G(s+2))$

$nCk(\{M1,M2,M3,M4\},k=2,s,dur=3,v_G(s+3))$

The first choice represents getting 2 GPU-enabled nodes, and completing in 2 time units with a start time s. The second choice captures all 2-combinations of nodes and represents running anywhere with a slower runtime of 3 time units. The STRL Generator combines these choices with a *max* operator, ensuring that the higher-value branch is chosen during optimization. A choice of $\{M1,M2\}$, for instance, will equate to the selection of the left branch, as visually represented in Fig. 3, if $v_G(s+2) > v_G(s+3)$. TetriSched subsequently combines such expressions for all pending jobs with a top-level *sum* operator to form the global optimization expression on each scheduling cycle.

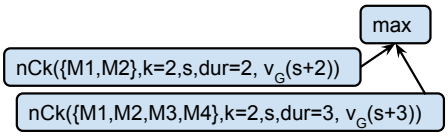


Figure 3. Soft constraint STRL example.

4.4 Deriving STRL from YARN jobs

This subsection explains and demonstrates how STRL is derived from YARN-managed jobs. YARN applications are written by supplying an ApplicationMaster (AM). It understands enough about the application structure to request resource containers at the right time, in the right quantity, in the right sequence, as well as with the right capabilities.[1] It is, therefore, fitting for such frameworks to supply the finer-granularity near-term information about submitted jobs to supplement coarser-granularity longer-term reservation information and trigger the corresponding STRL plugin to generate STRL expressions for managed job types. The AM then specifies whether it's an SLO or a best-effort job.

For example, suppose the GPU job in Sec. 4.3 (Fig. 3) arrives with a deadline=3 time units (Fig. 1). Then, its AM-specified RDL [7] expression would be:
Window(s=0, f=3, Atom(b=⟨16GB,8c⟩, k=2, gang=2, dur=3)), where the inner Atom() specifies a reservation request for a gang of 2 b-sized containers for a duration of 3 time units, and the Window() operator bounds the time range for the Atom() to $[0;3]$.[2] The resulting STRL expression then becomes

```
max(nCk({M1, M2, M3, M4},k=2, s=0, dur=3, v=1),
    max(nCk({M1, M2},      k=2, s=0, dur=2, v=1),
        nCk({M1, M2},      k=2, s=1, dur=2, v=1)))
```

The inner max composes all feasible start-time options for the job's preferred placement on GPU-nodes. The outer max composes all allocation options on preferred resources with a less preferred allocation anywhere ({M1, M2, M3, M4}). AM-specified performance slowdown factor is used to determine *dur*, while the range of start times comes from RDL-specified $[s;f]$. Estimates for different placement options can be learned by production cluster systems (e.g., Perforator [1]) over time for recurring production jobs. For some jobs, analytical models show accurate results across varying input sizes [38], and a number of systems have implemented a combination of performance modeling and profiling [12, 31]. Runtimes for unknown applications can be inferred from slowdown factors induced by heterogeneity [9, 10] coupled with initial estimates learned from clustering similar jobs (work in progress).

[1] MapReduce framework AM is a prime example of this.

[2] Please refer to [7] for complete RDL specification.

5. MILP Formulation

TetriSched automatically compiles pending job requests in the STRL language into a Mixed Integer Linear Programming (MILP) problem, which it solves using a commercial solver. The power from using the MILP formalism is twofold. First, by using the standard MILP problem formulation, we reap the benefit from years of optimization research that is built into commercial (and open-source) MILP solvers. Second, using MILP allows us to simultaneously schedule multiple queued jobs. Traditional schedulers consider jobs one at a time, typically in a greedy fashion that optimizes the job's placement. However, without any information about what resources other queued jobs desire, greedy schedulers can make suboptimal scheduling decisions.

TetriSched makes batch scheduling decisions at periodic intervals. At each scheduling cycle, it aggregates pending jobs using a STRL SUM expression, and solves the global scheduling problem. In our experiments, we aggregate all pending jobs, but TetriSched has the flexibility of aggregating a subset of the pending jobs to reduce the scheduling complexity. Thus, it can support a spectrum of scheduling batches of jobs from greedy one at a time scheduling to global scheduling.

Once it has a global STRL expression, TetriSched automatically compiles it into a MILP problem with a single recursive function (Algorithm 1). Recall that STRL expressions are expression trees composed of STRL operators and leaf primitives (Sec. 4.1). There are three key ideas underlying TetriSched's MILP generation algorithm.

First, we adopt the notion of binary indicator variables I for each STRL subexpression to indicate whether the solver assigns resources to a particular subexpression. Thus, our recursive generation function $\texttt{gen}(e, I)$ takes in an expression e and indicator variable I that indicates whether resources should be assigned to e. This makes it easy, for example, to generate the MILP for the MAX expression, which carries the semantics of "or". For a MAX expression, we add a constraint, where the sum of the indicator variables for its subexpressions is less than 1,[3] since we expect resources to be assigned to at most one subexpression.

Second, we find that the recursion is straightforward when the generation function returns the objective of the expression. At the top level, the return from the global STRL expression becomes the MILP objective function to maximize. Within the nested expressions, returning the objective also helps for certain operators, such as SUM and MIN. When compiling the SUM expression, the objective returned is the sum of the objectives returned from its subexpressions. When recursively compiling the MIN expression, objectives returned by its subexpressions help create constraints that implement MIN's "AND" semantics. We create a variable V, representing the minimum value, and for each subexpression we add a constraint that the objective returned is greater than V. As the overall objective is maximized, this forces all subexpressions of MIN to be at least V-valued.

Third, the notion of equivalence sets (Sec. 4.2) greatly simplifies the complexity of the MILP generation as well as the MILP problem itself. We group resources into equivalence sets and only track the *quantity* of resources consumed from each. Thus, we use integer "partition" variables to represent the number of resources desired in an equivalence set. We generate these partition variables at the leaf nCk and LnCk expressions, and use them in two types of constraints: demand constraints and supply constraints. Demand constraints ensure the nCk and LnCk leaf expressions get their requested number of resources, k. Supply constraints ensure that TetriSched stays within capacity of each equivalence set at all times. We discretize time and track integral resource capacity in each equivalence set for each discretized time slice.

5.1 MILP Example

Suppose 3 jobs arrive to run on a cluster with 3 machines {M1, M2, M3} (Fig. 4):

1. a short, urgent job requiring 2 machines for 10s with a deadline of 10s:
   nCk({M1, M2, M3}, k=2, start=0, dur=10, v=1)
2. a long, small job requiring 1 machine for 20s with a deadline of 40s:
   max(nCk({M1, M2, M3}, k=1, start=0, dur=20, v=1),
   nCk({M1, M2, M3}, k=1, start=10, dur=20, v=1),
   nCk({M1, M2, M3}, k=1, start=20, dur=20, v=1))
3. a short, large job requiring 3 machines for 10s with a deadline of 20s:
   max(nCk({M1, M2, M3}, k=3, start=0, dur=10, v=1),
   nCk({M1, M2, M3}, k=3, start=10, dur=10, v=1))

In this example, we discretize time in 10s units for simplicity and consider time slices 0, 10, 20, and 30. Note that, the only way to meet all deadlines is to perform global scheduling with plan-ahead. Without global scheduling, jobs 1 and 2 may run immediately, preventing job 3 from meeting its deadline. Without plan-ahead, we may either schedule jobs 1 and 2 immediately, making it impossible to meet job 3's deadline, or we may schedule job 3 immediately, making it impossible to meet job 1's deadline.

TetriSched performs global scheduling by aggregating the 3 jobs with a STRL SUM expression. It then applies our

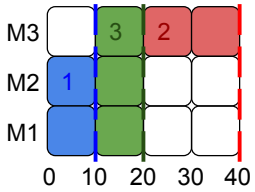


Figure 4. Requested job shapes, deadlines, and final order.

[3] Since the MAX expression itself may not be assigned any resources, depending on its indicator variable I, the constraint actually uses I instead of 1.

MILP generation function to the SUM expression, which generates 3 indicator variables, I_1, I_2, and I_3, that represent whether it is able to schedule each of the 3 jobs. It then recursively generates the variables and constraints for all jobs in the batch. Note that variables are localized to the subexpression where they are created, and constraints are added to a global *constraints* list. Thus, the algorithm names variables in the context of a subexpression, but, for clarity, in this example, we name variables more descriptively with globally unique names.

For the first job, represented by the above nCk expression, we create a partition variable $P_{1,s0}$, representing the amount of resources consumed by job 1 at time 0. Since there is only one partition in this example, $\{M1, M2, M3\}$, we omit the partition subscript. This partition variable is used in a demand constraint $P_{1,s0} = 2I_1$, indicating that job 1 needs 2 machines if it is scheduled (i.e., $I_1 = 1$). For the second job, we have a more complicated scenario with 3 options to choose from. We can start executing the job at time 0, 10, or 20. This is represented by the max expression, which is translated into 3 indicator variables corresponding to each of these options $I_{2,s0}$, $I_{2,s10}$, and $I_{2,s20}$.

Since we only want one of these options, the generation function adds the constraint $I_{2,s0} + I_{2,s10} + I_{2,s20} \leq I_2$. We use I_2 rather than 1 since the second job may not be selected to be run (i.e., $I_2 = 0$). For each of these 3 options, we recursively create partition variables $P_{2,s0}$, $P_{2,s10}$, and $P_{2,s20}$ and the corresponding constraints $P_{2,s0} = 1I_{2,s0}$, $P_{2,s10} = 1I_{2,s10}$, and $P_{2,s20} = 1I_{2,s20}$, representing the 1 machine that job 2 consumes in each option. For the third job, we have similar indicator variables $I_{3,s0}$ and $I_{3,s10}$, and partition variables $P_{3,s0}$ and $P_{3,s10}$, and constraints $P_{3,s0} = 3I_{3,s0}$, $P_{3,s10} = 3I_{3,s10}$, and $I_{3,s0} + I_{3,s10} \leq I_3$. After the recursion, we add supply constraints, representing the cluster capacity of 3 machines over time (see `genAndSolve` in Algorithm 1). For time 0, we add the constraint $P_{1,s0} + P_{2,s0} + P_{3,s0} \leq 3$. For time 10, we add the constraint $P_{2,s0} + P_{2,s10} + P_{3,s10} \leq 3$. Note that this constraint contains the term $P_{2,s0}$ because job 2 has a duration of 20s, and if it starts at time 0, it needs to continue running at time 10. For time 20, we add the constraint $P_{2,s10} + P_{2,s20} \leq 3$. For time 30, we add the constraint $P_{2,s20} \leq 3$. Solving this MILP produces the optimal solution (Fig. 4) of running job 1 immediately, running job 3 at time 10, and running job 2 at time 20.

6. Experimental Setup

We conduct a series of full system experiments to evaluate TetriSched's ability to schedule homogeneous and heterogeneous mixes of SLO and best effort jobs derived from production traces and from synthetic workloads. TetriSched is integrated into Hadoop YARN [35]—a popular open source cluster scheduling framework. We evaluate the performance of our proposed Rayon/TetriSched stack relative to the mainline YARN Rayon/CapacityScheduler(CS) stack.

```
gen: (expr, indicator var) → objective function
func gen(expr, I):
  switch expr:
    case nCk(partitions, k, start, dur, v)
      foreach x in partitions:
        P_x := integer variable  // Create partition variable
        for t := start to start + dur:
          // (Supply) Track resource usage
          Add P_x to used(x, t)
      // (Demand) Ensure this node gets k machines
      Add constraint Σ_x P_x = k * I
      return v * I  // Return value (if chosen - i.e., I = 1)
    case LnCk(partitions, k, start, dur, v)
      foreach x in partitions:
        P_x := integer variable  // Create partition variable
        for t := start to start + dur:
          Add P_x to used(x, t)
      Add constraint Σ_x P_x ≤ k * I
      return v * Σ_x P_x / k
    case sum(e_1, ..., e_n)
      for i := 1 to n:
        I_i := binary variable  // Create indicator variable
        f_i = gen(e_i, I_i)
      Add constraint Σ_i I_i ≤ n * I  // Up to n subexpr
      return Σ_i f_i
    case max(e_1, ..., e_n)
      for i := 1 to n:
        I_i := binary variable  // Create indicator variable
        f_i = gen(e_i, I_i)
      Add constraint Σ_i I_i ≤ I  // At most 1 subexpr
      return Σ_i f_i
    case min(e_1, ..., e_n)
      V := continuous variable  // Represents min value
      for i := 1 to n:
        f_i = gen(e_i, I)
        Add constraint V ≤ f_i  // Ensure V is min
      return V
    case scale(e, s)
      return s * gen(e, I)
    case barrier(e, v)
      f = gen(e, I)
      Add constraint v * I ≤ f
      return v * I
func genAndSolve(expr):
  I := binary variable  // Create indicator variable
  f = gen(expr, I)
  foreach x in partitions:
    for t := now to now + plan-ahead:
      // (Supply) Ensure usage ≤ avail resources
      Add constraint Σ_{P∈used(x,t)} P ≤ avail(x, t)
  solve(f, constraints)
```

Algorithm 1: MILP generation algorithm

Workload	SLO	BE	Unconstrained	GPU	MPI
GR_SLO	100%	0%	100%	0%	0%
GR_MIX	52%	48%	100%	0%	0%
GS_MIX	70%	30%	100%	0%	0%
GS_HET	75%	25%	0%	50%	50%

Table 1. Workload compositions used in results section.

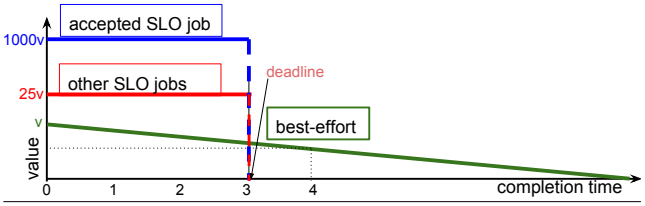


Figure 5. Internal value functions for SLO and BE jobs.

6.1 Cluster Configuration

We conduct experiments with two different cluster configurations: a 256-node real cluster (RC256) and an 80-node real cluster (RC80). For RC256, the experimental testbed [15] consists of 257 physical nodes (1 master + 256 slaves in 8 equal racks), each equipped with 16GB of RAM and a quad-core processor. RC80 is a subset of RC256 and is, therefore, a smaller, but similarly configured, 80-node cluster.

We maintain and use a single copy of YARN throughout an experiment, changing only the scheduler and workload for comparison experiments. We configure YARN with default queue settings and, generally, make YARN CS as informed and comparable to TetriSched as possible. First, we enable the Rayon reservation system. Second, we enable container preemption in CapacityScheduler, so that the scheduler can preempt running tasks to enforce Rayon capacity guarantees for reserved jobs. This gives a significant boost in terms of its ability to meet its capacity guarantees, particularly when the cluster is heavily loaded.

6.2 Workload Composition

Workloads are often composed of a mixture of job types as the jobs vary in their preferences and sensitivity to deadlines. Table 1 shows the workload compositions used for experiments reported in this paper.

6.2.1 Heterogeneity

For experiments with heterogeneous workloads, we use a set of job types that exemplify typical server-type and server-set preferences in production datacenters [9, 22, 30]. These preferences are captured with STRL, and corresponding STRL expressions are generated by the STRL Generator. For our heterogeneous mixes, we use three fundamentally different preference types: Unconstrained, GPU, and MPI.

Unconstrained: Unconstrained is the most primitive type of placement constraint. It has no preference and derives the same amount of benefit from an allocation of *any* k servers. It can be represented with a single "n Choose k" primitive, choosing k servers from the whole cluster serving as the equivalence set.

GPU: GPU preference is a simple and common [27, 30] example of a non-combinatorial constraint. A GPU-type job prefers to run each of k tasks on GPU-labeled nodes. Any task placed on a sub-optimal node runs slower (Fig. 3).

MPI: Rack locality is an example of a combinatorial constraint. Workloads such as MPI are known to run faster when all tasks are scheduled on the same rack. In our experiments, an MPI job prefers to run all k tasks on the same rack, while it is agnostic to which particular rack they are scheduled on. If the tasks are spread across different racks, all tasks are slowed down.

6.2.2 Deadline Sensitivity

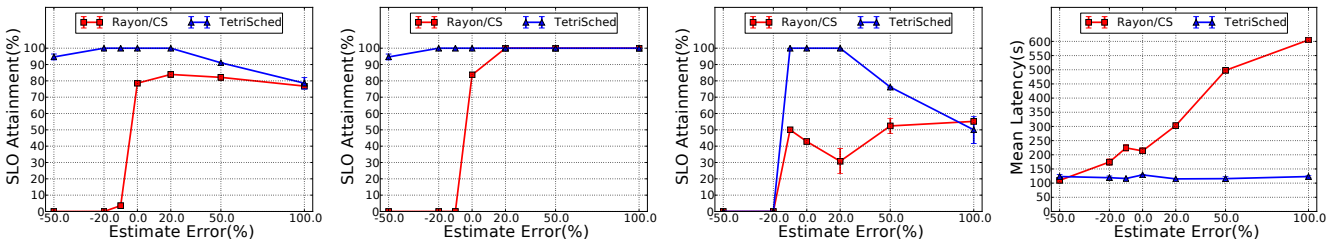
Our workloads are composed of 2 classes of jobs: Service Level Objective (SLO) jobs with deadlines and Best Effort (BE) jobs with preference to complete faster. An SLO job is defined to be *accepted* iff its reservation requested was accepted by the Rayon reservation system—used for both Rayon/CS and Rayon/TetriSched stacks. Otherwise, we refer to it as an SLO job without reservation (SLO w/o reservation). A job is defined as a best-effort(BE) job iff it never submitted a reservation request to Rayon. In our experiments, a value function $v(t)$ encodes the sensitivity to completion time and deadlines (Fig. 5). Best-effort $v(t)$ is a linearly decaying function with a starting value set to the same positive constant throughout all experiments. SLO $v(t)$ is a constant function up to a specified deadline, where the constant is 1000x the BE constant for accepted SLO and 25x for SLO w/o reservation, prioritizing them accordingly.

6.3 Evaluation metrics, parameters, policies

Throughout this paper, four main metrics of success are used: (a) accepted SLO attainment, defined as the percentage of *accepted* SLO jobs completed before their deadline; (b) total SLO attainment, defined as the percentage of *all* SLO jobs completed before their deadline; (c) SLO attainment for SLO jobs w/o reservation, defined as the percentage of SLO jobs w/o reservation completed before their deadline; (d) mean latency, defined as the arithmetic mean of completion time for best-effort jobs.

We vary two main experimental parameters: estimate error and plan-ahead. **Estimate error** is the amount of mis-estimation added to the actual runtime of the job. Positive values correspond to over-estimation, and negative mis-estimate corresponds to under-estimation. It exposes scheduler robustness to mis-estimation handling. **Plan-ahead** is the interval of time in the immediate future considered for deferred placement of pending jobs. Increased plan-ahead translates to increased consideration of scheduling jobs in time and generally improves space-time bin-packing. TetriSched cycle period is set to 4s in all experiments.

We experiment with four different TetriSched configurations (Table 2)) to evaluate benefits from (a) soft constraint awareness, (b) global scheduling, and (c) plan-ahead by having each of these features individually disabled (Sec. 7.2). TetriSched-NG policy derives benefit from TetriSched's soft



(a) all SLO jobs (b) SLO jobs w/reservations (c) SLO jobs w/o reservations (d) Best Effort latency

Figure 6. Rayon/TetriSched outperforms Rayon/CapacityScheduler stack, meeting more deadlines for SLO jobs (with reservations and otherwise) and providing lower latencies to best effort jobs. Cluster:RC256 Workload:GR_MIX

constraint and time-awareness, but considers pending jobs one at a time. It organizes pending jobs in 3 FIFO queues in priority-order: top priority queue with accepted SLO jobs, medium-priority with SLO jobs without a reservation, and low-priority with best-effort jobs. On each scheduling cycle, TetriSched-NG picks jobs from each queue, in queue priority order. TetriSched-NH policy disables heterogeneity-awareness at STRL generation stage by creating STRL expressions that draw k containers from only one possible equivalence set : the whole cluster. It uses the specified slowdown to conservatively estimate job's runtime on a (likely) sub-optimal allocation.

6.4 Workload Generation

We use a synthetic generator based on Gridmix 3 to generate MapReduce jobs that respect the runtime parameter distributions for arrival time, job count, size, deadline, and task runtime. In all experiments, we adjust the load to utilize near 100% of the available cluster capacity.

SWIM Project (GR_SLO, GR_MIX): We derive the runtime parameter distributions from the SWIM project [4, 5], which includes workload characterizations from Cloudera, Facebook, and Yahoo production clusters. We select two job classes (fb2009_2 and yahoo_1) with sizes that fit on our RC256 cluster. The GR_MIX workload is a mixture of SLO (fb2009_2) and BE (yahoo_1) jobs. The GR_SLO workload is composed solely from SLO jobs (fb2009_2) to eliminate interference from best-effort jobs.

Synthetic (GS_MIX, GS_HET): To isolate and quantify sources of benefit, we use synthetic workloads to explore a wider range of parameters. We evaluate our synthetic workloads on our smaller RC80 cluster. The GS_MIX workload is a mixture of homogeneous SLO and BE jobs. The GS_HET workload is a mixture of heterogeneous SLO jobs with varying placement preferences and homogeneous BE jobs.

TetriSched	TetriSched with all features
TetriSched-NH	TetriSched with No Heterogeneity (soft constraint awareness)
TetriSched-NG	TetriSched with No Global scheduling
TetriSched-NP	TetriSched with No Plan-ahead

Table 2. TetriSched configurations with individual features disabled.

7. Experimental Results

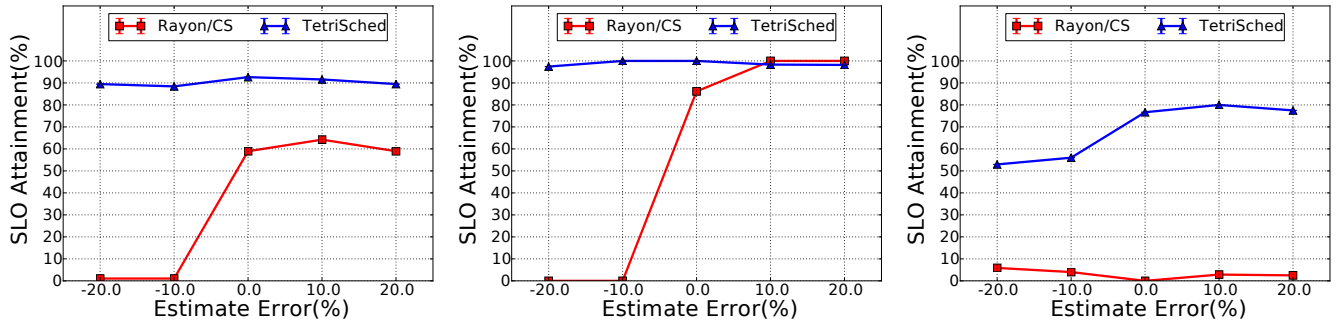
This section evaluates TetriSched, including its robustness to runtime estimate inaccuracy, the relative contributions of its primary features, and its scalability. The results show that TetriSched outperforms the Rayon/CapacityScheduler stack, especially when imperfect information is given to the scheduler, in terms of both production job SLO attainment and best effort job latencies. Each of TetriSched's primary features (soft constraints, plan-ahead, and global scheduling) is important to its success, and it scales well to sizable (e.g., 1000-node) clusters.

7.1 Sensitivity to runtime estimate error

Fig. 6 compares TetriSched with Rayon/CS on the 256-node cluster, for different degrees of runtime estimate error. TetriSched outperforms Rayon/CS at every point, providing higher SLO attainment and/or lower best effort jobs latencies. TetriSched is particularly robust for the most important category—accepted SLO jobs (those with reservations)—satisfying over 95% of the deadlines even when runtime estimates are half of their true value.

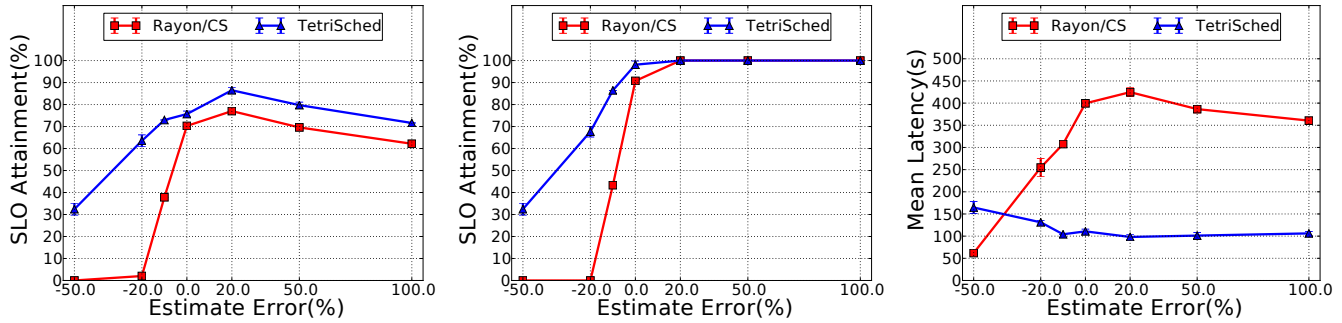
When job runtimes are under-estimated, the reservation system will tend to accept more jobs than it would with better information. This results in reservations terminating before jobs complete, resulting in transfer of accepted SLO jobs into the best-effort queue in Rayon/CS. Jobs in the best-effort queue then consist of a mixture of incomplete accepted SLO jobs, SLO jobs without reservations, and best-effort jobs. This contention results in low levels of SLO attainment and high best effort job latencies. In contrast, Rayon/TetriSched optimistically allows scheduled jobs to complete if their deadline has not passed,, adjusting runtime under-estimates upward when observed to be too low. It reevaluates the schedule on each TetriSched cycle (configured to 4s), adapting to mis-estimates by constructing a new schedule based on the best-known information at the time.

When runtimes are over-estimated, both schedulers do well for accepted SLO jobs. TetriSched satisfies more SLOs for jobs without reservations, because it considers those deadlines explicitly rather than blindly inter-mixing them, like the CapacityScheduler. Rayon/CS also suffers huge increases in best effort job latencies, because of increased pressure on the best-effort queue from two main sources: (1)



(a) all SLO jobs (b) accepted SLO jobs (c) SLO w/o reservation

Figure 7. Rayon/TetriSched achieves higher SLO attainment for production-derived SLO-only workload due to robust mis-estimation handling. Cluster:RC256 Workload:GR_SLO.



(a) all SLO jobs (b) accepted SLO jobs (c) latency

Figure 8. Synthetically generated, unconstrained SLO + BE workload mix achieves higher SLO attainment and lower latency with Rayon/TetriSched. Cluster:RC80 Workload:GS_MIX

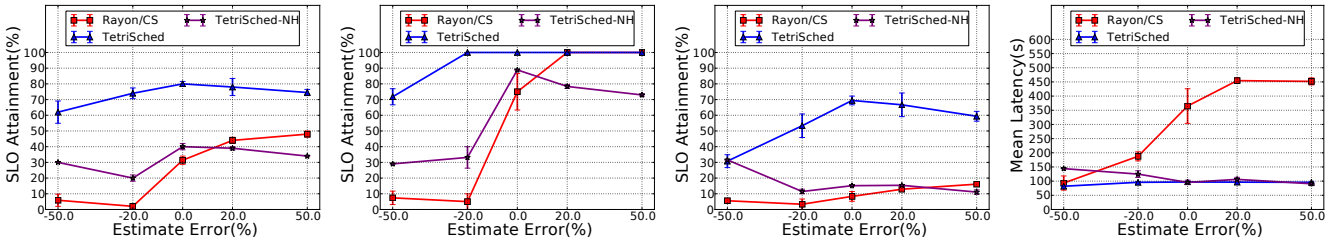
the number of SLO jobs without reservations increases with the amount of over-estimation; (2) the deadline information for any SLO jobs in the best-effort queue is lost, causing resources to be wasted on SLO jobs that cannot finish by their deadline. In contrast, TetriSched avoids scheduling such jobs. Additional resource contention arises from increased use of preemption. As over-estimate-based reservations are released early, temporarily available capacity causes more best-effort jobs to be started. But, these jobs often don't complete before the next SLO job with a reservation arises, triggering preemption that consumes time and resources.

To isolate the behavior of SLO jobs, without interference from best-effort jobs, we repeated the experiment with only SLO jobs; Fig. 7 shows the results. Now, the only jobs in the best-effort queue are (1) SLO jobs without reservations and (2) accepted SLO jobs with under-estimated runtimes. The results are similar, with Rayon/TetriSched achieving higher SLO attainment overall and maintaining ≈100% SLO attainment for accepted SLO jobs.

7.2 Sources of benefit

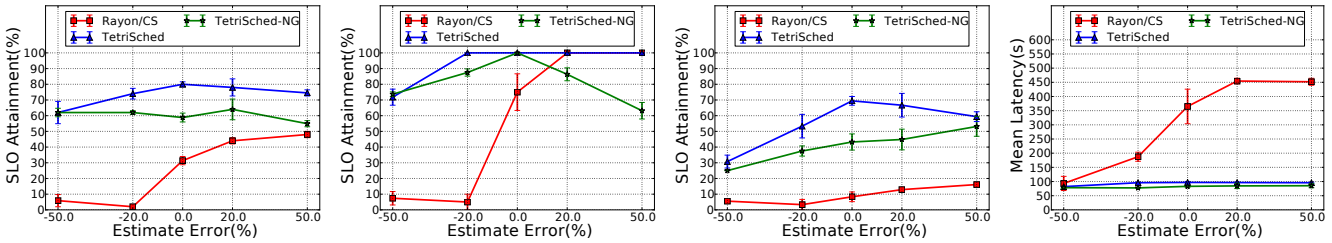
This section explores how much benefit TetriSched obtains from each of its primary features, via synthetically generated workloads exercising a wider set of parameters on an 80-node cluster. As a first step, we confirm that the smaller evaluation testbed produces similar results to those in Sec. 7.1 with a synthetic workload that is similar (homogeneous mix of SLO and best-effort jobs). As expected, we observe similar trends (Fig. 8), with TetriSched outperforming Rayon/CS in terms of both SLO attainment and best-effort latencies. The one exception is at 50% under-estimation, where TetriSched experiences 3x higher mean latency than Rayon/CS. The cause is that TetriSched schedules 3x more best-effort jobs (120 vs. 40), expecting to finish them with enough time to complete SLO jobs on time. Since TetriSched doesn't use preemption, best-effort jobs run longer, causing other best-effort jobs to accumulate queuing time, waiting for 50%-underestimated jobs to finish.

Soft constraint awareness. TetriSched accepts and leverages job-specific soft constraints. Fig. 9 shows that doing so allows it to better satisfy SLOs and robustly handle runtime estimate errors, for a heterogeneous workload mixture of synthetic GPU and MPI jobs combined with unconstrained best-effort jobs. This can be seen in the comparison of TetriSched to TetriSched-NH, which is a version of our scheduler with soft constraint support disabled. The key takeaway is that the gap between Rayon/TetriSched and TetriSched-NH is entirely attributed to TetriSched's support for soft constraints on heterogeneous resources. The gap is significant: 2-3x the SLO attainment (Fig. 9(a)). Disabling soft constraint support can even be seen reducing the performance of TetriSched-NH below Rayon/CS as over-estimation increases (Figures 9(a) and 9(b)). While both Rayon/CS and TetriSched-NH are



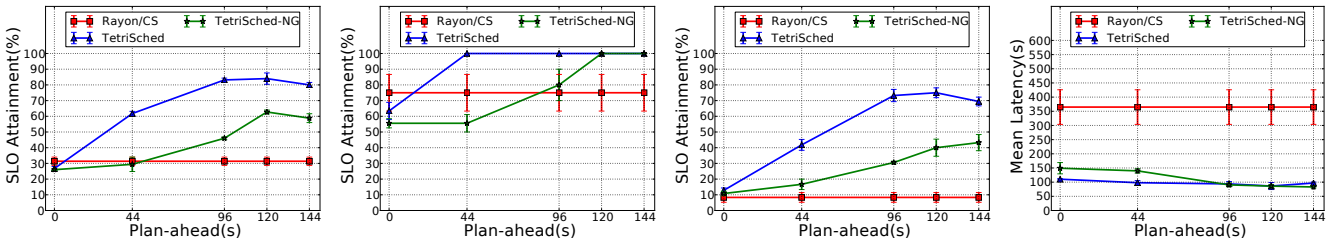
(a) all SLO (b) accepted SLO (c) SLO w/o reservation (d) BE latency

Figure 9. TetriSched derives benefit from its soft constraint awareness—a gap between TetriSched and TetriSched-NH. Cluster: RC80, Workload: GS_HET.



(a) all SLO (b) accepted SLO (c) SLO w/o reservation (d) BE latency

Figure 10. TetriSched benefits from global scheduling—a gap between TetriSched and TetriSched-NG. TetriSched-NG explores the solution space between Rayon/CS and TetriSched by leveraging soft constraints & plan-ahead, but not global scheduling. Cluster:RC80, Workload:GS_HET.



(a) all SLO jobs (b) accepted SLO jobs (c) SLO w/o reservation (d) latency

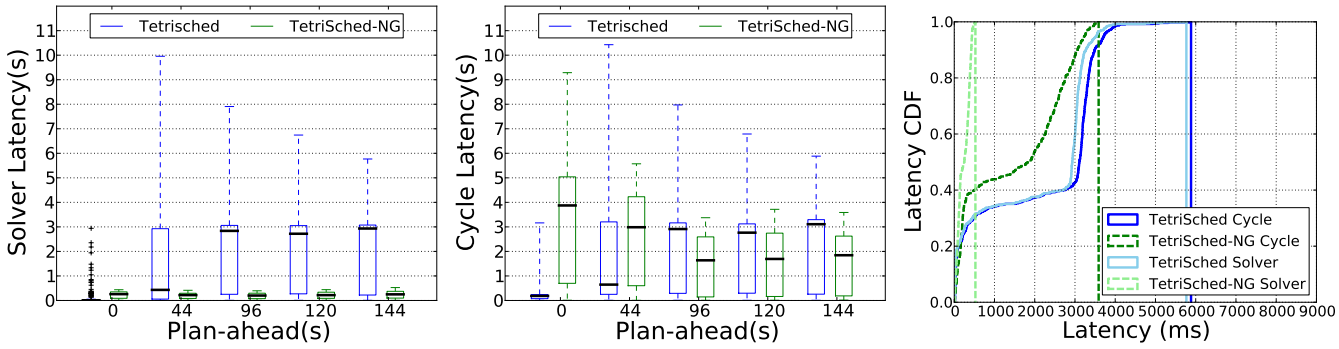
Figure 11. TetriSched benefits from adding plan-ahead to its soft constraint awareness and global scheduling. Cluster:RC80 Workload:GS_HET

equally handicapped by lack of soft constraint awareness, over-estimation favors Rayon/CS, as the job is started earlier in its reservation interval, increasing the odds of timely completion. TetriSched-NH on the other hand suffers from its lack of preemption when small best-effort jobs are scheduled at the cost of harder to schedule over-estimated SLO jobs. (Preemption in a TetriSched-like scheduler is an area for future work.)

Global scheduling. To evaluate the benefits (here) and scalability (Sec. 7.3) of TetriSched's global scheduling, we introduce TetriSched-NG, our greedy scheduling policy (Sec. 6.3). It uses TetriSched full MILP formulation, but invokes the solver with just one job at a time, potentially reducing its time complexity. Fig. 10 compares TetriSched with a version using the greedy policy, referred to as TetriSched-NG, finding that global scheduling significantly increases SLO attainment. Global scheduling accounts for the gap of up to 36% (at 50% over-estimate) between TetriSched and TetriSched-NG (Fig. 10(a)). TetriSched's global scheduling policy is particularly important for bin-packing heterogeneous jobs, as conflicting constraints can be simultaneously evaluated. We note that even TetriSched-NG outperforms Rayon/CS in both SLO attainment (Fig. 10(a)) and best-effort job latency (Fig. 10(d)), showing that greedy policies using TetriSched's other features are viable options if global scheduling latency rises too high.

Plan-ahead. Fig. 11 evaluates TetriSched and TetriSched-NG (with greedy scheduling instead of global), as a function of the plan-ahead window. (Note that the X-axis is plan-ahead window, not estimate error as in previous graphs.) When plan-ahead = 0 (i.e., plan-ahead is disabled), we see that, despite having soft constraint awareness and global scheduling, Rayon/TetriSched performs poorly for this heterogeneous workload. We refer to this policy configuration as TetriSched-NP (Sec. 6.3), which emulates the behavior of alsched [33]—our previous work. As we increase plan-ahead, however, SLO attainment increases significantly for TetriSched, until plan-ahead≈100s.

Summary. Fig. 9–11 collectively show that all three of TetriSched's primary features must be combined to achieve



(a) solver latency (b) cycle latency (c) CDF for plan-ahead=144

Figure 12. TetriSched scalability with plan-ahead.

the SLO attainment and best-effort latencies it provides. Removing any one of soft constraint support, global scheduling, or plan-ahead significantly reduces its effectiveness.

7.3 Scalability

Global re-scheduling can be costly, as bin-packing is known to be NP-Hard. Because TetriSched reevaluates the schedule on each cycle, it is important to manage the latency of its core MILP solver. The solver latency is dictated by the size of the MILP problem being solved, which is determined by the number of decision variables and constraints. Partition variables are the most prominent decision variables (Sec. 5) for TetriSched, as they are created per partition per cycle for each time slice of the plan-ahead window. Thus, in Fig. 12, we focus on the effect of the plan-ahead window size on the cycle (Fig. 12(b)) and solver (Fig. 12(a)) latencies. The cycle latency is an indication of how long the scheduler takes to produce an allocation decision during each cycle. The solver latency is the fraction of that latency attributed to the MILP solver alone. For the global policy, the solver latency is expected to dominate the total cycle latency for complex bin-packing decisions, as is seen in Fig. 12(c). The difference between cycle and solver latency is attributed to construction of the aggregate algebraic expression for pending jobs—overhead of global scheduling—and translating solver results into actual resource allocations communicated to YARN.

Fig. 12(b) reveals a surprising result: despite increasing the MILP problem size, increased plan-ahead can actually decrease cycle latency for the greedy policy. This occurs because scheduling decisions improve with higher plan-ahead (Sec. 7.2), reducing the number of pending jobs to schedule—another factor contributing to the size of the MILP problem. As expected, we can clearly see that the greedy policy (TetriSched-NG) decreases cycle and solver latency relative to global (TetriSched).

The combination of multiple optimization techniques proved effective at scaling TetriSched's MILP implementation to MILP problem sizes reaching hundreds of thousands of decision variables [34]. Optimizations include extracting the best MILP solution after a timeout, seeding MILP with an initial feasible solution from the previous cycle (Sec. 3.2), culling STRL expression size based on known deadlines, culling pending jobs that reached zero value, and most importantly, dynamically partitioning cluster resources at the beginning of each cycle to minimize the number of partition variables (Appendix A of [34])—all aimed at minimizing the resulting MILP problem size. Our companion TR [34] shows that TetriSched scales effectively to a 1000-node simulated cluster, across varied cluster loads, inter-arrival burstiness, slowdown, plan-ahead, and workload mixes. When we scale a simulation to a 10000-node cluster, running the GS_HET workload scaled to maintain the same level of cluster utilization as in Fig. 10), TetriSched exhibits a similar cycle latency distribution with insignificant degradation in scheduling quality. Even greater scale and complexity may require exploring solver heuristics to address the quality-scale tradeoff.

8. Related Work and Discussion

TetriSched extends prior research by addressing scheduling of diverse workloads on heterogeneous clusters via exploitation of estimated runtimes, plan-ahead with regular re-planning, and combinatorial placement preferences.

Handling Placement Preferences. The manner in which cluster schedulers consider placement preferences can be used to categorize them into four main classes, which may be termed `None`, `Hard`, `Soft`, and `Deferring`.

`None`-class schedulers don't model or understand placement preferences. Most such schedulers were designed assuming homogeneous infrastructures [22], focusing on load balancing and quantities of resources assigned to each job. This class includes schedulers using proportional sharing or random resource allocation for choosing placement candidates [25, 40]. Such schedulers fail to gain advantage from heterogeneous resources yielding opportunity costs when the benefits of getting preferred allocations are tangible.

`Hard`-class schedulers support specifying jobs with node preferences, but treat those specifications as requirements (i.e., as hard constraints). While such schedulers are heterogeneity-aware, their inflexible handling of placement preferences can be limiting, particularly when utilization is high. Based on prior work [30], this limitation contributes to noticeable performance degradation, increasing queuing de-

lays and causing jobs to unnecessarily wait for their specified preferences. This causes jobs to miss SLO targets, affects latency-sensitive jobs, and worsens utilization.

`Soft`-class schedulers address these issues by treating placement preferences as *soft constraints*. Notable examples include MapReduce [8], KMN [37], Quincy [18], and ABACUS [2]. To date, however, such schedulers have specialized for hard-coded types of preferences. Unlike the general-purpose approach of TetriSched, they hard-code support for handling specific placement preferences (e.g., data locality in Apollo [3]). Consequently, they lack the flexibility to adapt to new types of heterogeneity in both hardware and software. Similarly, CPU-centric observations have led to approaches based on greedy selection [9, 11, 24], hill-climbing [21, 23], and market mechanics [16]. An exception is Google's Borg [39], which supports soft constraints on diverse resources, but lacks support for estimated runtimes or time-based SLOs. Borg's scoring functions, built to scale, are bilateral, evaluating a job against a single node at a time. Borg also uses priorities as the mechanism for resolving contention rather than value-optimization as TetriSched does. Condor ClassAds [26] supports preferences, but is fundamentally bilateral, matching a single job to a single machine as Borg does. Consequently, both lack support for combinatorial constraints (generally, considered hard) and plan-ahead, provided by TetriSched.

`Deferring`-class schedulers, such as Mesos [17] and Omega [29], defer the complexity of reasoning about placement tradeoffs to second-level schedulers by exposing cluster state to pending resource consumers (via resource offers or shared state). The resource-offer approach (Mesos) has been shown to suffer livelocks due to hoarding to achieve preferred allocations (especially combinatorial constraints) [29], and neither addresses conflict resolution among preferences or time-based SLOs of different frameworks.

Comprehending space with Alsched. Alsched [33] sketched the idea of using utility functions to schedule jobs with soft constraints using a more primitive language to express them and a custom-coded bin-packing placement algorithm rather than TetriSched's general, automatically-generated MILP formulation. Alsched's support for soft constraints was limited to space only, as it completely ignored the time dimension. As such, it didn't support plan-ahead nor able to accommodate jobs with time-based SLOs. Alsched corresponds to TetriSched-NP in Sec. 6.3. Lastly, Alsched evaluation was with synthetic workloads in simulation only.

Comprehending time with Rayon. Rayon [7] supports SLO and best-effort mixes of complex datacenter workloads and stands out from most schedulers above in its comprehension of the time dimension—effected through the introduction of a rich language for resource capacity reservation. However, Rayon treats space as a scalar capacity (or multi-dimensional capacity); hence, TetriSched and Rayon are complementary as shown in this paper.

Fair Scheduling. Fairness has been a popular choice as an arbiter of resource contention, especially for academic clusters and federated resource pools with specified resource ownership proportions. However, mounting evidence from production clusters suggests that efficiency of scheduling significantly outweighs fairness considerations, which are only possible for omniscient observers. Further, a complete model for fairness in heterogeneous clusters and for workloads with soft constraints is an open problem. Max-min fairness assumes identical resources, DRF [13] considers capacity heterogeneity only, and the state-of-the-art constrained max-min fair (CMMF) scheduler [14] models only hard constraints, not the more expressive and, therefore, more complex space-time soft constraints.

9. Conclusion

TetriSched exploits time- and space-flexibility to schedule SLO and best-effort jobs on heterogeneous datacenter resources effectively. Job-specific resource preferences and temporal constraints are specified, together with reservation system-supplied deadlines and job runtime estimates, via its space-time request language (STRL). This information is leveraged to construct higher quality schedules by assigning the right resources to the right jobs, planning ahead which jobs to defer, and continuously re-evaluating to address new job arrivals and runtime mis-estimates. TetriSched performs global scheduling by batching multiple pending jobs and considering them for placement simultaneously, since constraints on diverse resources can arbitrarily conflict. Experiments with production-derived SLO and best-effort job mixes show higher SLO attainment and lower best-effort job latency, when compared to the state-of-the-art Hadoop YARN reservation and scheduling stack. Overall, TetriSched combines the features of general support for soft constraints in dynamically heterogeneous cluster space-time, combinatorial constraints, and plan-ahead in a scheduler that scales to hundreds of nodes. Integrated with the YARN reservation system, TetriSched is an appealing scheduling solution for heterogeneous clusters and cloud infrastructures.

10. Acknowledgments

We thank the member companies of the PDL Consortium (Avago, Citadel, EMC, Facebook, Google, HP Labs, Hitachi, Intel, Microsoft, MongoDB, NetApp, Oracle, Samsung, Seagate, Two Sigma, Western Digital) for their interest, insights, feedback, and support. This research is supported in part by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), by a Samsung Scholarship, by an NSERC Postgraduate Fellowship, and by the National Science Foundation under awards CSR-1116282, 0946825 and CNS-1042537, CNS-1042543 (PRObE [15])[4].

[4] http://www.nmc-probe.org/

References

[1] PerfOrator, 2015. http://research.microsoft.com/en-us/projects/perforator.

[2] AMIRI, K., PETROU, D., GANGER, G. R., AND GIBSON, G. A. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2000), ATEC '00, USENIX Association, pp. 25–25.

[3] BOUTIN, E., EKANAYAKE, J., LIN, W., SHI, B., ZHOU, J., QIAN, Z., WU, M., AND ZHOU, L. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 285–300.

[4] CHEN, Y., ALSPAUGH, S., AND KATZ, R. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. In *Proceedings of the VLDB Endowment* (2012), PVLDB.

[5] CHEN, Y., GANAPATHI, A., GRIFFITH, R., AND KATZ, R. The case for evaluating mapreduce performance using workload suites. In *MASCOTS* (2011).

[6] CHUN, B.-G. Deconstructing Production MapReduce Workloads. In *Seminar at: http://bit.ly/1fZOPgT* (2012).

[7] CURINO, C., DIFALLAH, D. E., DOUGLAS, C., KRISHNAN, S., RAMAKRISHNAN, R., AND RAO, S. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 2:1–2:14.

[8] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[9] DELIMITROU, C., AND KOZYRAKIS, C. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 77–88.

[10] DELIMITROU, C., AND KOZYRAKIS, C. Qos-aware scheduling in heterogeneous datacenters with paragon. *ACM Trans. Comput. Syst. 31*, 4 (Dec. 2013), 12:1–12:34.

[11] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 127–144.

[12] FERGUSON, A. D., BODIK, P., KANDULA, S., BOUTIN, E., AND FONSECA, R. Jockey: guaranteed job latency in data parallel clusters. In *Proc. of the 7th ACM european conference on Computer Systems* (2012), EuroSys '12, pp. 99–112.

[13] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant resource fairness: fair allocation of multiple resource types. In *Proc. of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)* (2011).

[14] GHODSI, A., ZAHARIA, M., SHENKER, S., AND STOICA, I. Choosy: max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 365–378.

[15] GIBSON, G., GRIDER, G., JACOBSON, A., AND LLOYD, W. PRObE: A thousand-node experimental cluster for computer systems research. *USENIX ;login: 38*, 3 (June 2013).

[16] GUEVARA, M., LUBIN, B., AND LEE, B. C. Market mechanisms for managing datacenters with heterogeneous microarchitectures. *ACM Trans. Comput. Syst. 32*, 1 (Feb. 2014), 3:1–3:31.

[17] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)* (2011).

[18] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 261–276.

[19] ISLAM, M., HUANG, A. K., BATTISHA, M., CHIANG, M., SRINIVASAN, S., PETERS, C., NEUMANN, A., AND ABDELNUR, A. Oozie: Towards a Scalable Workflow Management System for Hadoop. In *SWEET Workshop* (2012).

[20] JIA, Y., SHELHAMER, E., DONAHUE, J., KARAYEV, S., LONG, J., GIRSHICK, R., GUADARRAMA, S., AND DARRELL, T. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093* (2014).

[21] MARS, J., AND TANG, L. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2013), ISCA '13, ACM, pp. 619–630.

[22] MARS, J., TANG, L., AND HUNDT, R. Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity. *IEEE Computer Architecture Letters 10*, 2 (July 2011), 29–32.

[23] MARS, J., TANG, L., AND HUNDT, R. Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity. *Computer Architecture Letters 10*, 2 (July 2011), 29–32.

[24] NATHUJI, R., ISCI, C., AND GORBATOV, E. Exploiting platform heterogeneity for power efficient data centers. In *Fourth International Conference on Autonomic Computing (ICAC)* (2007), IEEE.

[25] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 69–84.

[26] RAMAN, R., LIVNY, M., AND SOLOMON, M. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)* (Chicago, IL, July 1998).

[27] REISS, C., TUMANOV, A., GANGER, G. R., KATZ, R. H., AND KOZUCH, M. A. Heterogeneity and dynamicity of clouds

at scale: Google trace analysis. In *Proc. of the 3nd ACM Symposium on Cloud Computing* (2012), SOCC '12.

[28] REISS, C., TUMANOV, A., GANGER, G. R., KATZ, R. H., AND KOZUCH, M. A. Towards understanding heterogeneous clouds at scale: Google trace analysis. Tech. Rep. ISTC-CC-TR-12-101, Intel Science and Technology Center for Cloud Computing, Apr 2012.

[29] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: flexible, scalable schedulers for large compute clusters. In *ACM Eurosys Conference* (2013).

[30] SHARMA, B., CHUDNOVSKY, V., HELLERSTEIN, J. L., RIFAAT, R., AND DAS, C. R. Modeling and synthesizing task placement constraints in Google compute clusters. In *Proc. of the 2nd ACM Symposium on Cloud Computing* (2011), SOCC '11, ACM, pp. 3:1–3:14.

[31] SHIVAM, P., BABU, S., AND CHASE, J. Active and accelerated learning of cost models for optimizing scientific applications. In *Proceedings of the 32Nd International Conference on Very Large Data Bases* (2006), VLDB '06, VLDB Endowment, pp. 535–546.

[32] SUMBALY, R., KREPS, J., AND SHAH, S. The Big Data Ecosystem at LinkedIn. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD.

[33] TUMANOV, A., CIPAR, J., KOZUCH, M. A., AND GANGER, G. R. alsched: algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proc. of the 3nd ACM Symposium on Cloud Computing* (2012), SOCC '12.

[34] TUMANOV, A., ZHU, T., KOZUCH, M. A., HARCHOL-BALTER, M., AND GANGER, G. R. Tetrisched: Space-time scheduling for heterogeneous datacenters. Tech. Rep. CMU-PDL-13-112, Carnegie Mellon University, Nov 2013.

[35] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., , LOWE, J., SHAH, H., SETH, S., SAHA, B., CURINO, C., O'MALLEY, O., RADIA, S., REED, B., AND BALDESCHWIELER, E. Apache hadoop yarn: Yet another resource negotiator. In *Proc. of the 4th ACM Symposium on Cloud Computing* (2013), SOCC '13.

[36] VENKATARAMAN, S., PANDA, A., ANANTHANARAYANAN, G., FRANKLIN, M. J., AND STOICA, I. The power of choice in data-aware cluster scheduling. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 301–316.

[37] VENKATARAMAN, S., PANDA, A., ANANTHANARAYANAN, G., FRANKLIN, M. J., AND STOICA, I. The power of choice in data-aware cluster scheduling. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2014), USENIX Association, pp. 301–316.

[38] VERMA, A., CHERKASOVA, L., AND CAMPBELL, R. H. Aria: Automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing* (New York, NY, USA, 2011), ICAC '11, ACM, pp. 235–244.

[39] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 18:1–18:17.

[40] WALDSPURGER, C. A., AND WEIHL, W. E. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1994), OSDI '94, USENIX Association.

[41] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 265–278.