Scheduling with Space-Time Soft Constraints In Heterogeneous Cloud Datacenters

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Alexey Tumanov

B.Sc., Computer Science & Mathematics, High Point University
M.Sc., Computer Science, York University

Carnegie Mellon University
Pittsburgh, PA

August, 2016

Keywords: cloud computing, cluster scheduling, heterogeneous datacenters, soft constraint-aware scheduling, space-time constraints, combinatorial constraints

To my family.

Abstract

Heterogeneity in modern datacenters is on the rise, in hardware resource characteristics, in workload characteristics, and in dynamic characteristics (e.g., a memory-resident copy of input data). As a result, which machines are assigned to a given job can have a significant impact. For example, a job may run faster on the same machine as its input data or with a given hardware accelerator, while still being runnable on other machines, albeit less efficiently. Heterogeneity takes on more complex forms as sets of resources differ in the level of performance they deliver, even if they consist of identical individual units, such as with rack-level locality. We refer to this as *combinatorial* heterogeneity. Mixes of jobs with strict SLOs on completion time and increasingly available runtime estimates in production datacenters deepen the challenge of matching the right resources to the right workloads at the right time.

In this dissertation, we hypothesize that it is possible and beneficial to simultaneously leverage all of this information in the form of declaratively specified *space-time soft constraints*. To accomplish this, we first design and develop our principal building block—a novel Space-Time Request Language (STRL). It enables the expression of jobs' preferences and flexibility in a general, extensible way by using a declarative, composable, intuitive algebraic expression structure. Second, building on the generality of STRL, we propose an equally general STRL Compiler that automatically compiles STRL expressions into Mixed Integer Linear Programming (MILP) problems that can be aggregated and solved to maximize the overall value of shared cluster resources.

These theoretical contributions form the foundation for the system we architect, called TetriSched, that instantiates our conceptual contributions: (a) declarative soft constraints, (b) space-time soft constraints, (c) combinatorial constraints, (d) orderless global scheduling, and (e) in situ preemption. We also propose a set of mechanisms that extend the scope and the practicality of TetriSched's deployment by analyzing and improving on its scalability, enabling and studying the efficacy of preemption, and featuring a set of runtime mis-estimation handling mechanisms to address runtime prediction inaccuracy.

In collaboration with Microsoft, we adapt some of these ideas as we design and implement a heterogeneity-aware resource reservation system called Aramid with support for ordinal placement preferences targeting deployment in production clusters at Microsoft scale. A combination of simulation and real cluster experiments with synthetic and production-derived workloads, a range of workload intensities, degrees of burstiness, preference strengths, and input inaccuracies support our hypothesis that leveraging *space-time soft constraints* (a) significantly improves scheduling quality and (b) is possible to achieve in a practical deployment.

Acknowledgments

First and foremost, I am deeply grateful to my advisor, Greg Ganger, for empowering and encouraging me to "do great things". The extent of unconditional support I received from Greg, the mentorship, the coaching— all transcended academic matters and included matters of my personal growth as well. Thank you, Greg, I now understand why we will be forever conflicted on all future paper submission reviews. Go team!

I would like to thank my thesis committee members, Prof. Greg Ganger (chair), Dr. Michael A. Kozuch, Prof. Mor Harchol-Balter, and Prof. Ion Stoica, for their collaboration, encouragement, and valuable feedback over the years of this dissertation work. Your unwavering belief in the successful outcome and impact of this work were motivating, refreshing, and inspiring. Many thanks to Prof. Garth Gibson and Prof. Majd F. Sakr for their keen interest in my research, for the many research conversations we've had over the years, and for their mentorship and guidance when I assisted with teaching the Advanced Cloud Computing course. It's an experience that will stay with me and will inform my future course instruction and development. Special thanks to Garth for PRObE [24], which enabled an amazing long-term hands-on access to bare metal. It was pivotal to the many projects I completed. Thanks to the Carnegie Mellon administration for allowing, sponsoring, and supporting partial deployments of PRObE and the Data Center Observatory (DCO) on their premises.

I am grateful to the three brilliant groups of researchers I interned with over the six years of my Ph.D: (1) Jean-Philippe Martin, Chris Rossbach, and Michael Isard at Microsoft Research Silicon Valley (circa 2011); (2) John Wilkes, Michael Abd-El-Malek, Walfredo Cirne, and the whole Omega team at Google (circa 2012); (3) Carlo Curino, Ishai Menache, Sriram Rao, Chris Douglas, Subru Krishnan and the whole CISL/MSR Redmond team at Microsoft, Inc. Thanks for your guidance, developing my taste for research ideas that are specifically relevant to industry, for your mentorship, and the generally great experience working with you.

I would like to thank one of my first PDL collaborators, James Cipar, for his energy, unwavering willingness to help, positive outlook on life, the universe, and everything, and, generally, being a role model for a successful Ph.D. student in Systems. I would also like to acknowledge Jim Cipar as well as Timothy Zhu, Jun Woo Park, and Angela Jiang for collaborating with me on cluster scheduling at various points of my time at Carnegie Mellon. The energy of our group projects, the adrenaline rush of the many conference deadlines, and the eventual successes we all shared are all too fresh in my memory to imagine ever forgetting.

There's a special place in my heart for the members of the Parallel Data Lab (PDL)— the group of people I feel the strongest professional affiliation with. The truly collaborative spirit of PDL, the open nature of group research, the ad hoc hallway conversations are the qualities of the environment I will seek and carry with me wherever I go. Thanks to my cube mates: Michelle Mazurek and Elie Krevat—for helping me see the value and attraction of work-life balance, Ilari Shafer for his brisk energy,

millennial colloquialisms, infinite positivity, and seemingly boundless supplies of snacks. Thanks to Raja Sambasivan for being the senior student passing the baton of PDL responsibilities to the next generation of PDLers and being the invaluable source of LaTeX magic. Thanks to Lianghong Xu for collaborating with me on several early projects, some of which he led to successful publication in esteemed venues. Thanks to the more recent wave of students who brighten up the environment with their fresh productivity, fresh perspective, and fresh ideas: Aaron Harlap, Angela Jiang, Rajat Kateja, and Henggang Cui. And, of course, the list of PDLers would never be complete without the mention of Karen—the PDL Headmistress, Joan Digney—for holding the fort of aesthetics in all professional communications, the research support staff: Mitch Franzos, Xiaolin Zang, Michael Stroucken, and, more recently, Chad Dougherty, Dr. Chuck Cranor, and Jason Boles.

I would like to express sincere gratitude to the members of the SAFARI group and Prof. Onur Mutlu. I am grateful to Onur for his genuine interest in and appreciation for my research ideas that straddled the disciplines of Computer Systems and Computer Architecture. Through two courses and a research project with Onur, I developed a deep appreciation for Onur's outstanding work ethic, perspective, and research vision. I am also grateful to the many student members of the SAFARI research group: Chris Fallin—I could always count on his insightful and deeply intellectual feedback and conversation; Gennady Pekhimenko—thanks for being a friend and setting an example for work ethic, and creation of collaborative project opportunities; Vivek Seshadri—thanks for the insightful research conversations; and, of course, thanks to other SAFARI students whom I interacted with over the years—there are too many to list here!

I thank the member companies of the PDL Consortium (Actifio, Avago, American Power Conservation (APC) Corporation, Citadel, EMC, Emulex, Facebook, Fusion-IO, Google, Hewlett Packard (HP) Labs, Hitachi, Huawei, Intel, Microsoft, MongoDB, NEC (formerly Nippon Electric Company) Labs, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, sTec (formerly Simple Technology), Symantec, Two Sigma, VMware, 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 a Natural Sciences and Engineering Research Council of Canada (NSERC) Canada Graduate Scholarship (NSERC CGS-D3), and by the National Science Foundation under awards CSR-1116282, 0946825 and CNS-1042537, CNS-1042543 (PRObE [24]).[1].

Last, but certainly not least, I would like to acknowledge the unconditional love and tremendous support that helped me start, continue, and complete this enormous undertaking from my wife Olena, my daughters Ekaterina and Marina—my Godsent angels, my sister Svitlana, and my parents: Nadiya and Alexander P. Tumanov. This Ph.D. dissertation simply would not have happened without you.

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Industry has embraced the benefits of consolidating various workloads on shared cluster infrastructures. Now, large clusters are shared by Big Data analytics batch jobs, High Performance Computing (HPC) simulations, machine learning (ML) model-training applications, web search, long-running front-end services, etc. Cluster framework diversity is evident and will continue to rise as new frameworks (e.g., Caffe [35] for deep learning) join the ranks of recent ML, graph, and data analytics frameworks like Spark, GraphLab, Hive, Giraph, MXNet, and Petuum. Heterogeneity in hardware is crucial to achieving efficiency, as different jobs and job phases run differently on different types and sets of servers. Some workloads also derive additional benefit from specialized interconnects, such as 2D torus overlays for reconfigurable FPGA fabric instances used to score web documents for Bing search [53], or Infiniband interconnects for latency critical High Performance Computing (HPC) and other bulk synchronous parallel (BSP) jobs. Cluster hardware heterogeneity will also continue to grow, as evidenced by the increase in Amazon AWS instance heterogeneity exposed to the user over the recent years. The combination of consolidation with increasing heterogeneity is a recent phenomenon for datacenters and presents a new challenge for cluster resource management systems. This creates a new scheduling problem: matching the right resources to the right workloads at the right times.

1.1 Background

When dealing with homogeneous resources, traditional schedulers are effective; their focus is on allocating quantities of resources to jobs. But, heterogeneity adds the additional consideration of *which* resources (i.e., where the job is placed). While some schedulers ignore heterogeneity, most support job-specific constraints on which resources the job is willing to use. Using constraints to specify the best choice for a job can avoid receiving a sub-optimal placement. But, since best-choice resources may not be immediately available, using *hard* constraints to indicate which resources would be best can lead to high queuing delay [59, 66] and lower utilization—both despite the availability of other acceptable machines.

Using hard constraints to specify preferred resources over-constrains the scheduler. With sufficient over-provisioning, it is acceptable, but a different approach is needed to simultaneously realize (i) the utilization benefits of consolidation and (ii) the efficiency benefits of heterogeneity. This dissertation proposes the use of *soft constraints* [66] in cluster resource space to convey which resources are better (i.e., job-specific placement preferences) together with an indication of how much better. Armed with such information, a scheduler can explore the trade-offs involved, attempting to optimize the matching of resources to jobs.

It is common for production cluster workloads to have time considerations as well as spatial preferences. Timing flexibility is a second degree of freedom that has emerged. For example, the vast majority of jobs in clusters at Microsoft [10] can be classified into two categories: completion-time SLO driven and latency sensitive with varying degrees of urgency of their execution. The recurrent nature of most of these jobs can be leveraged [10, 20] to inform the scheduler about their estimated runtimes on various hardware configurations. For some workloads [20, 76], analytical models can be used to extend these estimates to a range of relevant input sizes.

Thus, workloads in heterogeneous datacenters have both preferences and requirements on where (space), when (time), and how (space-time shape) they run. Yet, little of this knowledge

and flexibility is leveraged in cluster schedulers today. Space-time preferences, which we call *space-time soft constraints*, are either partially or fully ignored, inflexibly considered mandatory, or hard-coded in scheduling logic, limiting extensibility and scope. This dissertation proposes fully embracing this two-dimensional flexibility, by providing the necessary building blocks for the space-time soft constraints to be expressed, their benefits—quantified, and the knowledge—leveraged to create better cluster schedules than state-of-the-art approaches that waste compute resources or human time.

The challenge of scheduling for heterogeneous clusters is exacerbated by the fact that the solution space is often *combinatorial*. Rack-local jobs can run on any k of machines on the same rack on any rack. Data-intensive analytics applications can choose any k-subset from a set of nodes that store their data [72]. Failure-sensitive applications must survive up to k' simultaneous failures, prompting no more than k' tasks allocated per failure domain (e.g., PDU, rack, or data-center) and so on. Our insight, however, is that this combinatorial explosion of choices can not only be succinctly represented, but also *leveraged* to produce more efficient cluster schedules compared to schedulers that either don't consider these placement considerations or inflexibly treat them as required.

Cluster scheduling research goes back decades. But the increasing levels of constraint-inducing static and dynamic heterogeneity (discussed in §2) combined with the new scale of datacenters, resource and workload types present a new multi-faceted challenge. A successful solution must be able to:

1. capture resource and workload heterogeneity in a general and expressive way that need not change with new types of hardware and software;
2. flexibly support both simple and *combinatorial* constraints found in datacenters today in terms of where to run (space);
3. flexibly support temporal considerations, such as queuing delay, completion time SLOs, calendared jobs, and estimated runtimes, i.e. when and how long to run (time); and

4. efficiently allocate available resources to maximally benefit running jobs, increasing their heterogeneous metrics of interest.

1.2 Thesis Statement

This dissertation confirms the effectiveness of a heterogeneity-aware resource allocation mechanism simultaneously aware of both spatial and temporal resource preferences in achieving better resource allocations than alternate approaches without such knowledge. Specifically:

Support for explicit expression, quantification, and exploitation of declarative space-time soft constraints for scheduling of heterogeneous jobs in dynamic, heterogeneous datacenters is (a) possible and (b) beneficial for effective assignment of heterogeneous resources.

The dissertation provides the following evidence in support of this thesis statement.

1. We characterize real workloads from a large (≈ 12500 node) Google cluster cell, focusing on its heterogeneity and scheduling constraints to motivate the problem and as an existence proof of complex forms of dynamic heterogeneity in large-scale production clusters today.
2. We demonstrate that it is possible to succinctly represent highly complex placement preference structures by designing and implementing support for a space-time request language (STRL)—an algebraic expression language for declaratively expressing heterogeneous objectives in a general fashion. STRL's generality further accommodates forms of heterogeneity in both hardware and software that do not yet exist, as long as they fit the mathematical model of relational algebra we use as base.
3. We demonstrate that exploitation is possible by building a scheduler, called TetriSched, and evaluate the effectiveness of scheduling mechanisms that have access to and ability to comprehend such placement preferences.
4. To confirm that support for explicit expression, quantification, and exploitation is benefi-

cial, we perform extensive real system experimentation and simulation analysis. Our results indicate that TetriSched is indeed more effective in resource allocation than a variety of alternative options, including: (i) schedulers that treat preferences as hard constraints, (ii) schedulers that ignore preferences, (iii) real schedulers that have limited, hard-coded support for certain specialized types of preferences, such as YARN's data locality preferences.

5. To extend the practicality of the fundamental building blocks proposed, we additionally demonstrate that:

   (a) TetriSched robustly handles a range of job runtime mis-estimation that is observed in real production clusters

   (b) TetriSched natively supports preemption "in situ", as it leverages its STRL and MILP Compiler to simultaneously consider jobs for (i) preemption, (ii) placement, and (iii) deferral.

   (c) TetriSched supports a more typical, greedy, single job at a time scheduling policy to accommodate environments where per-job scheduling latency is critical. Indeed, the composability of STRL makes it possible to schedule any number of pending jobs (from one to all) per invocation of the MILP solver. This is a practical step towards explicit management and control of MILP instance complexity generated by the MILP Compiler.

1.3 Dissertation Focus

This dissertation focuses on the inefficiencies of resource management in dynamic, heterogeneous clusters. We make two key high-level observations:

1. (a) the degree and (b) complexity of heterogeneity is on the rise in both production cluster resources and production jobs, creating a feedback loop, whereby more diversity in

workloads incentivizes more heterogeneity in hardware and vice versa.

2. existing scheduling systems are not equipped to handle this rapidly spiraling resource allocation complexity. This leads to significant inefficiencies in cluster resource management. As a direct result, clusters are over-provisioned, causing lower utilization (and, therefore, lower ROI), and/or cluster users don't get predictable performance or any control over when, where, and how their jobs are executed.

In this dissertation, we concern ourselves with the *design, development, analysis, and evaluation of the necessary primitives, algorithms, mechanisms, and systems to enable declarative space-time soft constraints in dynamic heterogeneous cluster resource management.*

Achieving this enables cluster scheduling systems to leverage inherent flexibility in cluster jobs' placement needs and execution timeframe. It improves the quality of resource allocation, by allowing the scheduler to make better bin-packing decisions, increasing cluster ROI. Lastly, it gives power users more control over the spatial and temporal details of their workload execution. We achieve this by building on a foundation of 2 key building blocks:

1. space-time request language (STRL), described in §3
2. Mixed Integer Linear Programming (MILP) compiler for STRL, described in §4

Together, these building blocks enable a list of novel TetriSched contributions, outlined in §1.4.

1.4 Contributions

This dissertation makes the following contributions.

1.4.1 Conceptual Contributions

Declarative Soft Constraints

With rising levels and types of heterogeneity, existing scheduling systems struggled to cope with managing the resulting complexity in a general fashion. Ad-hoc solutions typically involved stat-

ically partitioning cluster resources (e.g., into separate job submission queues based on resource types), hard-coding heterogeneity-awareness for specific application types (e.g., data locality, rack locality, anti-affinity), ignoring it, or deferring the complexity to higher-level frameworks (e.g., as done by two-level schedulers, such as Mesos). We address this challenge in a general manner to ensure that the resulting solution stands the test of time, as new types of heterogeneity inevitably emerge. We thus introduce a notion of soft constraints as a way of capturing and quantifying a, potentially, exponential number of resource assignments in a declarative fashion. Soft constraints, intuitively and most succinctly, define a mapping function on the superset of cluster resources to $\mathbb{R}$. We differentiate between cardinal and ordinal soft constraints. Cardinal soft constraints are mathematical functions mapping the elements from the cluster resource superset to real numbers. Ordinal soft constraints define a preference relation on the domain of cluster resource superset. The biggest challenge with declarative soft constraints is defining such a function (or a preference relation) in a succinct fashion, avoiding enumeration of exponentially many placement options. We achieve this with the design and development of a space-time request language (STRL), described in §3.

Space-time soft constraints

Building on declarative soft constraints, we introduce the notion of space-time soft constraints to reason about resource allocation as a two-dimensional construct defined by (a) a set of resources used (space) and (b) an interval of time it's intended to be used for (time). Informally, we refer to these two-dimensional constructs as "space-time rectangles" throughout this dissertation.[1] The shift from soft constraints in space only to 2D space-time soft constraints is fundamental. The latter achieves strictly superior expressivity and an extra degree of freedom for resource allocation. To bridge these two notions conceptually, consider spatial soft constraints as space-time soft constraints specified for infinity. As shown experimentally in §7, having a temporal component to soft constraints is essential to achieving the highest effectiveness. Space-time soft constraints,

[1]Naturally, when visualized, these rectangles are not necessarily contiguous in space.

formally, are functions that map such two-dimensional, spatiotemporal cluster resource shapes to $\mathbb{R}$. Similarly to soft constraints above, a less expressive alternative is for these functions to be a preference relation instead.[2]

Combinatorial Constraints

Combinatorial constraints are an important category that should be accommodated explicitly. We define *simple* constraints as a function that maps a quantity drawn from one resource subset (however specified) to $\mathbb{R}$. Examples of such simple constraints include a preference to run k tasks on machines with a GPU (or any other attribute). Most prior HPC literature on various labeling schemes and algebraic operations on those schemes to describe cluster resources fundamentally reduces down to the same idea: they specify an equivalence set of resources, a subset of resources that satisfy some predetermined criteria. Simple constraints then may operate on this set (however specified) by mapping some quantity of the set to a scalar value. It is, of course, possible for a simple constraint to be a point function.

In contrast, *combinatorial* constraints cannot be specified by operating on a single subset alone. Combinatorial constraints involve multiple subsets and can only be specified by using higher order operators on a *set of resource subsets*. Rack locality is one simple example of a combinatorial constraint. To declaratively specify a rack locality constraint, one must operate on a set of resource subsets (each of which is a rack) and map a desired quantity of k containers from each subset to a scalar value (e.g., a boolean, if the constraint is hard). A higher level operator then declares ANY of those outcomes as admissible. Anti-affinity and relaxed anti-affinity (whereby no more than k' tasks are requested per resource subset) are good examples of combinatorial constraints, natively supported by STRL (§3). In fact, the generality and expressivity of STRL is such that it enables sweeping the continuum of options between a strict rack locality constraint (all k tasks on one rack of many) and a strict anti-affinity constraint (no more than 1

[2]Preference relation is defined on X as a subset of $X \times X$ (the Cartesian product of X with self), where an element $(x, y) \in X \times X$ implies $x \succeq y$ (x is preferred to y).

task per rack) in its entirety. Everything in between is expressible in STRL, and this continuum exemplifies combinatorial constraints.

Orderless global scheduling

As resource requests and their associated spatiotemporal placement preferences are specified and encoded in STRL, they can be aggregated and composed into a single high-level algebraic expression. This dissertation describes an algorithm to *automatically* compile any STRL expression into a canonical form of Mixed Integer Linear Programming (MILP). Solving this problem has an important property of considering all declaratively specified constraints simultaneously. This obviates the need to decide on the order in which jobs should be considered for placement. As we discuss in §5.2.2, heterogeneity has the undesired effect of breaking the nice properties of associativity and commutativity that make homogeneous cluster scheduling much easier. The order in which jobs are considered for placement starts to matter more with more constraints and more resource types over which those constraints are specified. In fact, it can be shown that it is possible for ANY order of jobs in a set of job order permutations to produce a space-time schedule that is suboptimal, compared to simultaneous consideration of all jobs and their constraints for placement. We explore the benefits of global scheduling in §7.4.2.

In Situ Preemption

Leveraging orderless global scheduling, we propose a mechanism to perform preemption simultaneously with considering jobs for placement. We refer to this as "in situ preemption". Drawing on the expressive power of STRL combined with its automatic translation to MILP, it is possible to consider all or a subset of running jobs for preemption simultaneously with all or a subset of pending jobs for placement—both specified as STRL expressions, aggregated, and translated to a single canonical instance of MILP. As the cost of preempting running jobs intuitively contributes negative value, maximizing the combined STRL expression will have the desired effect

of minimal necessary preemptions to achieve maximal possible value from the set of available resources.

1.4.2 Theoretical Contributions

Space-Time Request Language (STRL)

STRL is the fundamental building block that is a necessary condition for all of the conceptual contributions listed above. It enables the ability to declaratively specify a fundamentally new class of placement constraints succinctly. It directly addresses the main challenge of centralized, heterogeneity-aware scheduling in heterogeneous contexts—ability to succinctly specify and quantify the combinatorial explosion of placement options. We explain why this is hard in §2. STRL itself is designed as a collection of language primitives, consisting of language operands and operators. Its "n choose k" language primitive serves as the enabling foundation for STRL's succinct method of reducing exponentially many options of identical value to a single language primitive. Even combinatorial constraints, where a mapping has to be defined on a set of resource subsets, STRL can express this in O(number of subsets), even though we draw k units from each of the subsets, resulting in combinatorially many options to consider.

STRL to MILP compilation

An algorithm that automatically compiles any STRL expression to MILP is essential to the intended generality of our design and implementation. Furthermore, reducing a set of independent resource requests along with a set of preemptible jobs to a single canonical MILP problem instance is an enabling mechanism for orderless global scheduling and in situ preemption.

1.4.3 Systems Artifacts

1. TetriSched—we've instantiated the ideas above in a system called TetriSched. TetriSched is architected to operate independently as a scheduling policy server. It was, in fact, used

in standalone mode for the numerous simulation experiments. TetriSched was also integrated with Hadoop YARN to leverage YARN's ResourceManager for cluster resource and job life cycle management, interacting with cluster resource consumers, monitoring job progress, and servicing cluster resource heartbeats delivered by individual NodeManagers. TetriSched plugs in as an independent scheduler, replacing YARN's default CapacityScheduler. We operated this integrated YARN/TetriSched system on a real 256-node cluster at Carnegie Mellon, serving hundreds of jobs with the scheduler core configured to operate on the scheduling cycle of 4 seconds. TetriSched was able to scale despite having to continuously solve MILP problems to service resource requests. We describe a set of engineering optimizations in §5.

2. Aramid—a system we built in collaboration with Microsoft to instantiate a subset of TetriSched ideas in a system aimed at production cluster deployment. Aramid implements the notion of heterogeneity-awareness and ordinal placement constraints on top of YARN's stock reservation system called Rayon [10]. The key systems primitive we introduce through Aramid work is the "variable space-time capacity guarantee queue" construct. It vectorizes the previously scalar handling of resource capacity in YARN, coupling it with the variable capacity guarantees introduced by Rayon. The variable space-time capacity guarantee queue effectively implements space-time soft constraints in a widely deployed, open source resource management framework, with open sourcing efforts underway.

1.5 Dissertation Overview

We start with background and motivation in Chapter 2 by offering a brief taxonomy of heterogeneity (§ 2.1). This chapter explains the magnitude of the challenge of scheduling for heterogeneous resource environments. It also makes precise the intended meaning of the overloaded

term “heterogeneity”. We further categorize various known approaches to handling placement preferences engendered by heterogeneity (§2.2). We further describe a host of mechanisms that can be used to quantify placement options and conclude the chapter with the discussion of the temporal considerations of cluster scheduling. Space-time soft constraints—the key conceptual contribution of our work, rests on the assumption that runtime estimates for resource requests are available. (We do not, however, assume that they are accurate. Section 5.2.5 describes a set of mechanisms to robustly handle runtime mis-estimation. In §2.4.1, we focus on categorizing the various mechanisms that have been used to predict job runtimes.

In Chapter 3, we proceed to describe the design of the main theoretical contribution of this work—the Space-Time Request Language. We introduce the notion of “equivalence sets” in §3.1.1. Equivalence sets capture a subset of resources deemed indistinguishable relative to a given job. We emphasize throughout this work that equivalence sets are dynamically constructed as a property of the job, viewing the set of resources through its own lens. We develop the intuition for STRL in §3.2, formally specify the primitives and operators, and provide several common examples. To anchor this discussion in the context of a widely deployed YARN resource management framework, we offer an example of how STRL expressions can be derived from the combination of YARN resource requests and Rayon reservation requests. We conclude this chapter with the discussion and formal specification of STRL primitives that enable “in situ preemption”.

In Chapter 4, we discuss the second theoretical contribution of our dissertation work —the Mixed Integer Linear Programming formulation and the algorithm that creates it automatically for any given STRL expression. In §4.3, we separately draw the reader’s attention to MILP generation for STRL’s preemption primitives. We build up the intuition for the automatic MILP problem construction as a recursive expression tree descent, creating value in the leaves of the tree and modifying the flow of this value in its non-leaf vertices with the end goal of maximizing the overall value of the STRL expression. Handling supply and demand capacity constraint

construction is also discussed.

Chapter §5 serves as an architecture chapter. There we bring together the expressive power of STRL and MILP into a system architecture that instantiates our conceptual contributions. End-to-end system architecture overview can be found in Fig. 5.1 and described in §5.1. The operation of the TetriSched core is detailed in §5.2, which expands on a set of architectural building blocks that comprise TetriSched, including plan-ahead (§5.2.1), global scheduling (§5.2.2), in situ preemption (§5.2.3), and mechanism for runtime mis-estimation handling (§5.2.5). We conclude with a brief overview of TetriSched's integration with YARN.

In Chapter 6, we detail the experimental setup, system policies chosen for comparison, workload composition, cluster composition, and metrics of success for both a set of simulation experiments and a set of real cluster experiments. Some tunable workload parameters (such as load and slowdown) are also introduced in this chapter.

We present empirical evaluation results in support of our thesis statement in Chapter 7. The structure of this chapter is as follows. We start by demonstrating the benefit of soft constraints relative to constraint handling systems that treat preferences as required or ignore them. We follow up with a deep dive under the hood (§7.1.1) to explain how TetriSched derives these performance benefits and which of the success metrics ultimately contribute to the overall value. The second major evaluation section §7.2 demonstrates the effect of adding the second, temporal dimension to our space-time soft constraints. We detail specific workload characteristics, such as workload inter-arrival burstiness, that particularly benefit from TetriSched's ability to plan ahead. We transition to a set of real cluster experiments with a transitional sensitivity experiment that aims to quantify the effect of runtime mis-estimation on the quality of TetriSched's placement decisions. Based on the results of these experiments, we make a surprising discovery that TetriSched is robust to runtime mis-estimation of up to 2x of the job's ground truth runtime.

We transition to a set of experiments carried out on a real cluster with TetriSched integrated with YARN in §7.4. The focus of §7.4.1 is to validate the earlier sensitivity study conducted in

simulation. Sensitivity to mis-estimation is shown to be far greater in the state-of-the-art YARN's default stack as compared to YARN integrated with TetriSched. We break down the sources of benefit in §7.4.2, focusing on soft constraint awareness, global scheduling, and plan-ahead. Finally, we explore the scalability of TetriSched in §7.4.3.

In Chapter 8, we describe the instantiation of the subset of our conceptual contributions in a different system, called Aramid. The significance of this effort is in targeting a production cluster environment a priori. We overview the extent and types of heterogeneity observed the target cluster environments in §8.1. Section 8.2 details the high-level scheduler components and their interaction in Aramid. Aramid's reservation definition language with our heterogeneity extensions is presented in Section 8.3. There, we focus on describing and developing intuition for our main contributions to the language, namely the heterogeneity-awareness extensions to the language primitive and the introduction of ordinal soft constraints. Finally, Section 8.4 presents empirical evaluation results that demonstrate that the instantiation of a subset of our conceptual contributions had a significant effect on the number and the aggregate size of resource reservations, evaluated at the scale of a 2700-node production cluster.

We conclude with Chapter 9, where we start by sharing some of the main high-level takeaways and lessons learned in §9.1. We discuss the scope of TetriSched's effectiveness as well as limitations in §9.2 and capture some of the intended follow-up work in §9.3.

Chapter 2

Background and Motivation

2.1 Taxonomy of Heterogeneity

Cluster jobs have become very heterogeneous [56, 57, 59]. This trend is expected to continue and, in fact, gain further momentum [29]. New specialized frameworks of execution will continue to emerge. While previously the number of job types in a given enterprise cluster was usually low, and resources could be statically partitioned among separate clusters, it has become common practice to consolidate compute and data resources to take advantage of shared data and statistical multiplexing properties. As a consequence, cluster resources themselves are becoming increasingly diverse, to tailor to the needs of diverse workloads they support. Thus, hardware specialization is also on the rise, making 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 [59]. Static heterogeneity is on the rise and is expected to continue to increase [29, 35] . We refer to such node specialization as *static* heterogeneity—diversity rooted in the static parameters of cluster resources.

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*. Their jobs consist of tasks with input and output dataflow relationships. Tasks read their data from distributed local storage or cached in memory and pass their outputs to downstream intermediate tasks. Their efficiency depends in part on the locality of tasks to data they consume. From such jobs' perspective, a set of machines becomes heterogeneous, when viewed through the lens of data locality. Machines with desired data become preferred to machines without it.

Presence or co-location with interfering workloads [15, 16] 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.

Cluster operators and site reliability engineers at companies that maintain their own production clusters may also assign machine roles to different parts of the cluster. Examples include labeling a fraction of a cluster as "persistent" to ensure that critical application components (e.g., YARN application masters) are placed on machines with a priori lower probability of failure, handover, or CPU frequency downthrottling. These assigned attributes are (a) often dynamic in nature, and (b) follow neither from hardware (static) properties nor from software (dynamic) properties, contributing to the need for the scheduler to understand dynamically changing resource attributes to make efficient placement decisions.

Resource churn also contributes to the dynamicity of the pool of resources. At production scale, at any given point in time, it is likely for some nodes to be upgraded, maintained, and generally, acquire new capabilities or dynamic attributes. Unlike static heterogeneity, resource

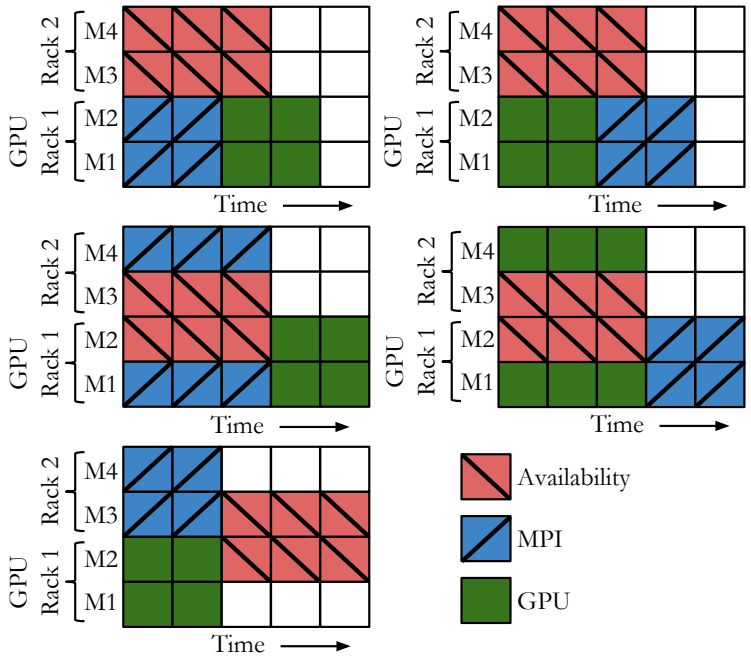


Figure 2.1: Five potential schedules for 3 jobs. Each grid shows one potential space-time schedule, with machines along the rows and time units along the columns. Each job requests 2 servers, and its allocation is shown by filling in the corresponding grid entries. The cluster consists of 2 racks each 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).

churn and general node failures create a transient effect on resource accounting. As we attempt to keep track of resource capacity for resource subsets defined by machine attributes, we must make sure that scheduling mechanisms employed are robust to (a) transient and (b) frequent changes to those resource sets. This is a qualitative departure from siloed, vertical partitioning of resources either into separate management domains or into separate queues, based on the assumption that machine attributes do not change. In addition to per-node attribute upgrades, sets of nodes can acquire new connectivity through network backbone upgrades, deployment of redundant, low latency interconnects, such as Infiniband, and deployment of specialized high-performance interconnect overlays, such as the FPGA Torus interconnects used for Microsoft's Catapult project.

Combinatorial heterogeneity: Finally, as exemplified by HPC and Machine Learning applications with tightly-coupled communicating tasks, different identically sized *subsets* of machines may influence job execution differently. Some subsets (e.g., rack-local collections of k machines [66], data-local samples of data servers [72]) may speed up the execution or improve application QoS, while others may degrade it. An interesting new trend in data analytics contributes additional examples where applications must choose a subset of the data to operate on [72], such as (a) approximate query processing, (b) machine learning, and (c) erasure coded storage applications. All subsets, however, could be "good enough" to enable applications to run. We refer to such heterogeneity as *combinatorial*. Jobs that prefer all k tasks to be simultaneously co-located (e.g. MPI job in Fig. 2.1) in the same locality domain of the many locality domains 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. A distinguishing characteristic of combinatorial heterogeneity is its presence in what could otherwise be deemed a completely homogeneous cluster with identical machines.

To gain an intuitive understanding of combinatorial heterogeneity, it helps to think of success metrics of performance that can only be evaluated, when a given set of resources is known, and cannot be evaluated based on quantity alone. Putting it in functional terms, the domain of the preference or value function is no longer the amount of resource allocated (on the x-axis). Rather it is the set of all possible subsets of resources. The function, in principle, maps the relevant subsets of interest to the performance success metric of choice (typically a scalar value in $\mathbb{R}$). Given the fact that the cardinality of a superset is exponential in the number of elements in the set, it becomes evident that a concise definition of this function is a necessary condition for any solution addressing combinatorial heterogeneity.

TetriSched captures all such placement considerations succinctly with STRL described in §3.

2.2 Handling Placement Preferences

As alluded to in the previous section, the rising levels of heterogeneity give rise to placement preferences in consolidated cluster environments. Fundamentally, a preference is either an explicit or implicit ordering operator on the superset of available resources. Due to the complexities of (a) identifying the preference order, (b) quantifying the degree of preference, and (c) providing the mapping for a meaningful fraction of the preference function domain in a succinct fashion, the manner in which cluster schedulers consider placement preferences varies significantly. We categorize schedulers into four main categories, with respect to their ability to comprehend, express, and leverage placement preferences. We refer to them as `None`, `Hard`, `Soft`, and `Deferring`.

`None`-class schedulers (as in Fig. 2.2) don't model or understand placement preferences. Most such schedulers were designed assuming homogeneous infrastructures, 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 [51, 78]. Such schedulers fail to gain advantage from heterogeneous resources yielding opportunity costs when the benefits of getting preferred allocations are tangible. This class of schedulers also includes schedulers that, by design, concern themselves with fairness. It includes the Hadoop Fair scheduler [1] and, more recently, the work done by Ali Ghodsi et al. on dominant resource fairness [22] and the extension of that work to support hierarchical queues [5].

A large class of schedulers we refer to as `Hard` filled this semantic gap by defining resource description schema and expressions on top of it. It enabled the specification of preferred types of resources, which were treated as required and inflexibly honored by the scheduler. While this class of schedulers was heterogeneity-aware (in contrast to `None`), the inflexible handling of placement considerations as *hard* constraints is a limiting factor. Based on prior work [59] as well as our own experiments (Fig. 2.2), this limitation contributes to noticeable performance degradation, increasing queueing delays and causing jobs to unnecessarily wait for their specified

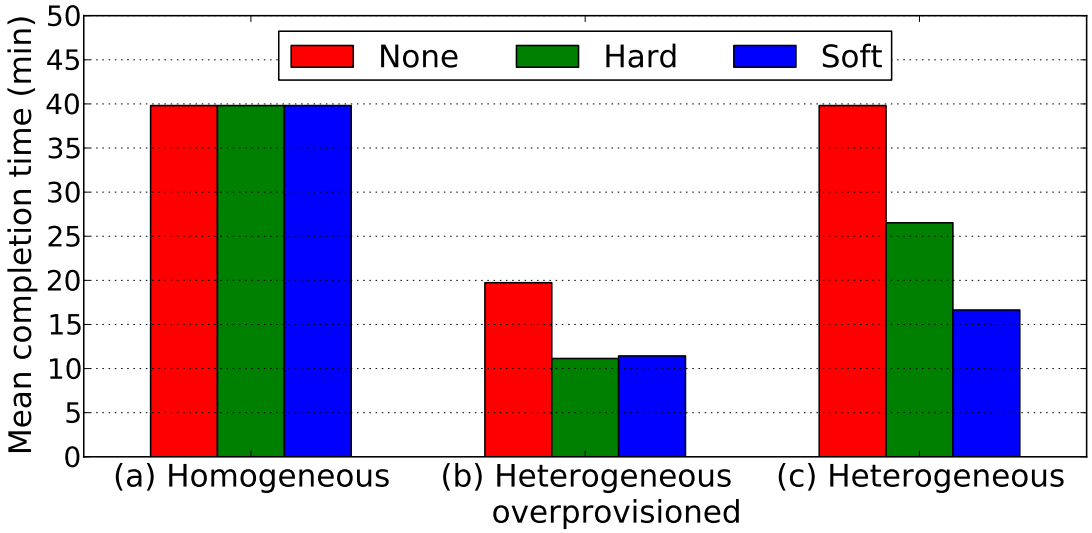


Figure 2.2: Exploiting placement preference flexibility is crucial in heterogeneous infrastructures. Each set of three bars are simulation results for jobs submitted to one of three sizable clusters: (a) a homogeneous cluster of 1000 generic machines; (b) an over-provisioned heterogeneous cluster of 1500 machines, wherein any machine can run any job but the best-match machine (e.g., one with a GPU) does so with 50% lower execution time; (c) a (not-over-provisioned) heterogeneous cluster of 1000 machines. For each, the three bars represent three classes of schedulers: `None` corresponds to schedulers that assume all machines are equivalent (no placement preferences are considered); `Hard` refers to schedulers that treat placement preferences as required; `Soft` corresponds to TetriSched, which is aware of placement preferences and their magnitudes (e.g., 50% faster), considering fallbacks as needed. The results show that realizing the benefits of heterogeneity requires that schedulers be aware of placement preferences, unlike when scheduling for homogeneous clusters. Further, unless over-provisioned, the scheduler must flexibly consider trade-offs associated with placement preferences to achieve the best performance.

preferences, possibly indefinitely. In turn, jobs miss SLO targets, latency-sensitive applications suffer loss of quality due to increased queueing delays, and cluster utilization worsens.

`Soft`-class schedulers address these issues by treating placement preferences as *soft constraints*. Notable examples include MapReduce [13], KMN [73], Quincy [32], and ABACUS [3]. To date, however, such schedulers have specialized for very specific types of preferences. Unlike the general-purpose approach of TetriSched, they hard-code support for handling specific placement preferences (e.g., input data locality). Consequently, they lack the flexibility to adapt to new types of heterogeneity in both hardware and software. Similarly, efforts primarily focused on dif-

ferences in processor properties have led to approaches based on greedy selection [15, 17, 49], hill-climbing [45, 46], and market mechanics [26]. An exception is Google's Borg [77], which comprehends soft constraints of diverse resources, but does not appear to use time estimates, and employs static priority as the mechanism for both preemption and resolving contention unlike TetriSched's universal use of value optimization. Condor ClassAds [55] supports behavior akin to "soft constraints"; however, it is fundamentally bilateral, matching a single job to a single machine. It also lacks support for combinatorial constraints and gang scheduling, as provided by TetriSched. Gang scheduling support helps TetriSched avoid hoarding (and associated potential for deadlock) needed by other systems (e.g., Mesos [29], YARN [71]) to achieve the same effect. [25] addresses the challenge of scheduling for heterogeneity, but focuses on the impact of poor decisions induced by heterogeneity. We believe the work by Guevara et al. is, therefore, complementary, as it can be used to estimate degrees of preference and slowdown factors, contributing to the construction of STRL expressions (§ch:strl).

`Deferring`-class schedulers, such as Mesos [29] and Omega [58], defer the complexity of reasoning about placement tradeoffs to the hosted application frameworks 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) [58], and both approaches leave unresolved the issue of conflict resolution among preferences of different frameworks. TetriSched addresses these issues with a flexible language for soft constraints in cluster space-time.

2.3 Quantifying Placement Tradeoffs

It is inevitable for placement preferences to conflict. Moreover, the nature of this conflict is no longer a matter of resource capacity. With the three types of heterogeneity—all prevalent in production datacenters today—placement preferences overlap in complex ways. To make informed scheduling decisions, one needs to arbitrate conflicting preferences. To do so, placement trade-

offs must be quantified. When resources are over-provisioned and sufficient to meet all placement preferences in the worst case, the scheduling problem is easy (Fig.2.2(b)). Over-provisioning is expensive, however, and unpredictable demand for certain types of resources may cause temporal imbalances and result in higher wait times. A scheduler that understands how to handle resource contention flexibly (`Soft`) is desired (Fig. 2.2(c)). To inform its arbitration, however, such a scheduler must be able to quantify placement tradeoffs.

Several approaches to quantifying placement tradeoffs have been used. First, *fairness* is a popular choice in academic clusters, where proportional sharing of resources relative to the agents' resource contributions and/or importance is a primary concern. Note that fairness should, indeed, be viewed as just a way to quantify tradeoffs when resources are contended. Second, some large-scale Internet companies and public clouds use *revenue* as a mechanism for quantifying placement tradeoffs. Either real or virtual currency has been used [30, 31]. For such environments, the scheduler quantifies relative benefits of placement options based on the aggregate revenue that can be achieved. Third, *performance* can be used as a tie-breaker. The scheduler chooses resource assignments such that a contended resource is given to the job whose performance will benefit the most from it.

We utilize the fact that *utility*—an economic theory mechanism assigning scalar value to allocation outcomes—is expressive enough to represent most of these approaches to quantifying placement tradeoffs. Throughout our thesis work, we use utility as a general way of quantifying tradeoffs.

Fairness. Fairness is a popular choice as an arbiter of resource contention, especially for academic clusters and federated resource pools with specified resource ownership proportions. Previous work has used several models, including max-min fairness [4], dominant resource fairness [22], and the state-of-the-art constrained max-min fair (CMMF) scheduler [23]. But, we do not yet have a complete model for fairness in heterogeneous clusters. Max-min fairness assumes identical resources, DRF [22] considers capacity heterogeneity only, and CMMF [23] models

only hard constraints, which are not sufficient for preferences (See Fig. 2.2(c)).

In the scope of this dissertation, we consider a more general case of heterogeneity, described in §2.1, as well as support for soft placement preferences. For such arbitrarily heterogeneous pools of resources and workload mixes with simple and combinatorial placement preferences, the notion of fairness remains *undefined*. Indeed, intuitively, given a mix of coconut and bananas, fairness will not be achieved by proportionally dividing each pool of fruit, if some people are allergic to coconut, some prefer bananas, and some are indifferent, as long as they get some fruit. It would be achieved only if (a) all fruit were the same or (b) all persons were indifferent. Defining fairness calls for a more nuanced modeling approach that captures and normalizes this heterogeneity. We believe that defining fairness for environments with soft and combinatorial constraints is an interesting question for future work that uses utility to quantify fairness across diverse users with declaratively specified preference structures over a heterogeneous pool of resources. For this reason, we *purposefully* leave fairness outside the scope of this dissertation work.

Revenue. Another option is to allocate resources to the highest bidder and let the market decide resource bid prices through supply and demand. The scheduler then allocates resources in a way that maximizes revenue. Currency can be real or virtual with or without restrictions on per-user or per-job budget allocations. Prior work on this suggests that charging users real currency makes the system incentive-compatible [31].

Performance. Yet another option is to allocate resources in a manner that achieves maximum aggregate performance. This can be done only in the cases when all jobs' performance metrics are comparable and can be aggregated. Examples include HPC clusters that primarily focus on job completion times, such as Hadoop/Spark analytics and MPI batch jobs. In such cases, the scheduler can quantify tradeoffs using the aggregate performance metric as the objective to optimize. Note that the mix of jobs is heterogeneous, and scheduled over a (statically or dynamically) heterogeneous mix of hardware. But, the performance metrics are homogeneous.

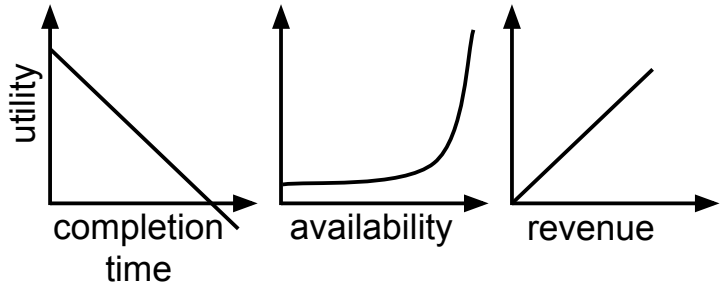


Figure 2.3: Mapping heterogeneous objectives to utility.

Utility functions. Utility has been widely used to quantify placement tradeoffs in the scheduling literature [36, 37, 39, 40, 41, 52, 62, 79]. It is attractive due to its generality, accommodating performance, availability, revenue, etc. Indeed, Fig. 2.3 illustrates how performance metrics, such as completion time and availability, as well as revenue naturally map to utility. Utility in the form of a virtual currency can also be used to achieve fairness via allocation of budget and rate of its replenishment [31]. Doing so for heterogeneous environments we target, however, remains an open research question.

Quantifying tradeoffs in TetriSched. TetriSched can use *any* of the mentioned ways of quantifying placement tradeoffs. In §3, we capture placement options by mapping them to a user-defined and user-interpreted scalar value. It could be a measure of performance, revenue, or utility. It must be understood, that utility functions , exemplified in Fig. 2.3, serve as a layer of indirection with the following benefits. For job mixes with completion time as a measure of performance, they enable differentiation of user urgency (steeper slope), user importance (higher budget), and job deadline SLO. For job mixes with heterogeneous user objectives, utility serves as the unifying currency allowing the scheduler to quantify placement tradeoffs across jobs with incomparable objectives (e.g., completion times and availability).

Similarly to [20, 52], we choose time-based utility functions depicted in Fig. 2.4 as our placement tradeoff quantification mechanism. To quote Lee et al [41], "there is a legitimate concern that users may not be willing to provide this level of detail in a real-life job submission setting." However, the authors' previous work [42] had shown that such information is easily obtainable.

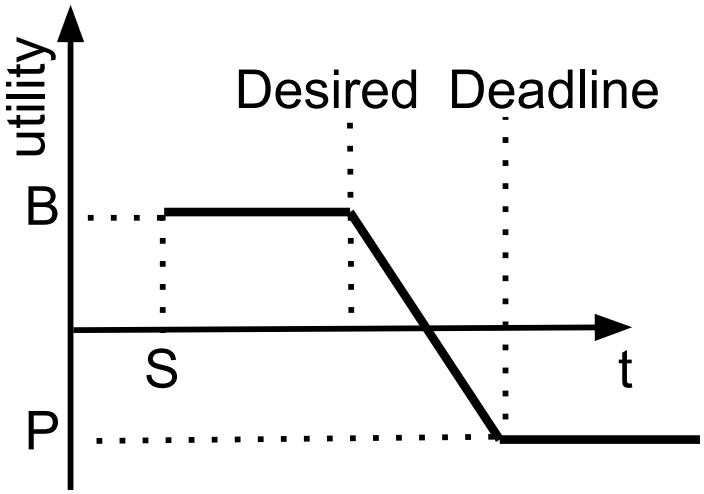


Figure 2.4: User objective function with respect to time. Temporal user objectives are modeled as time-based user-defined utility functions (uduf). Budget B is the maximum utility possible for this job– to be awarded if the job is started or completed by the Desired time (depending on whether it is start- or completion-time oriented). Potential utility declines if the Desired target is not met, and penalty P (negative utility) accrues for failure to meet the Deadline. The earliest start time S enables the expression of calendared jobs.

In cases where it's not, TetriSched's STRL Generator provides reasonable defaults. Additionally, we show that TetriSched is robust to user runtime estimate misrepresentation in §7.4 and §7.3.

Thus, we borrow utility functions from prior work as *one* of several ways of quantifying heterogeneous placement preferences. User-specified utility functions serve as an optional high-level interface to the scheduling system. We build on it by providing a general algebraic expression language for capturing simple and combinatorial placement preferences. It serves as the main interface to the core scheduler, obviating the need for it to change with new types of heterogeneity. It is important to emphasize that the building blocks we propose and the mechanisms that use them are oblivious to the actual quantification method of choice.

2.4 Temporal considerations

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 [10, 16, 17, 18, 20, 76] has demonstrated the feasibility of generating such estimates.

Given job runtime estimates, recent scheduling work [20, 70] shows that accurate job runtime predictions can be exploited to significant benefit when dealing with workload and resource diversity. Below we discuss three ways of exploiting submission-time job runtime information.

Resource type assignment in heterogeneous clusters. As discussed above, datacenter resources, today, are increasingly heterogeneous; consolidation of multiple workloads onto a shared infrastructure does not remove the efficiency benefits of resource specialization. Rather than having separate clusters for each machine type, modern clusters consist of a mixture of machines with different sizes (e.g., huge-memory machines), special accelerators (e.g., GPUs or FPGAs), and ages (due to incremental cluster growth and refresh). As previously stated in §2.1 , various asymmetries may also arise, such as in interconnect topology (e.g., per-rack switches) and data locality (e.g., cached executables and/or input data).

Jobs will be affected differently by specific machine assignments. Some jobs may execute largely the same on any machines, a common assumption in historical schedulers; for others, only certain assignments may be acceptable. A growing category of jobs, however, are those with *soft constraints* [66, 69, 70]; such jobs can accept many assignments but will run faster or more robustly given specific assignments (e.g., on a GPU-equipped machine or with all tasks on the same rack). Maximizing cluster effectiveness in the presence of jobs with such *soft constraints* requires careful scheduling.

Scheduling jobs with soft constraints is more effective when job runtimes are known [6, 70, 82]. Naturally, the scheduler may assign the *preferred* resources, provided that they are available when a new job is first considered for scheduling. But, if they are not, the scheduler may need to decide whether assigning less-preferred resources immediately is better than waiting for preferred resources to become available (and/or freeing them via preemption). A scheduler can make a better, more informed decision if it knows how long the wait would be and how long the new job would run on alternative resource subsets.

In the absence of runtime knowledge, some schedulers [71, 82] use *delay scheduling*—waiting for a small, pre-configured amount of time for preferred resources to become available and falling back to alternatives if they remain occupied. Of course, sometimes jobs wait when they should not, and sometimes they don't wait long enough, because the wait time is a configuration parameter applied in all cases.

Packing deadline-oriented jobs with latency-sensitive jobs. Cluster workloads are increasingly a mixture of business-critical production jobs and best-effort engineering/analysis jobs. The production jobs are often workflows submitted periodically by automated systems [33, 63] to process data feeds, refresh models, and publish insights. These jobs often consume significant resources (e.g., many servers and tens of TBs of data), run for hours, and have strict completion deadlines(i.e., completion-time Service Level Objectives, or SLOs). The best-effort jobs, such as exploratory data analytics and software development/debugging, while lower priority, are often latency-sensitive. Schedulers can more effectively order jobs when given their runtimes, simultaneously increasing SLO attainment for production jobs and reducing average latency for best-effort jobs [10, 70]. Such information enables tighter packing of jobs in cluster "space-time". By carefully exploiting available slack between production job submission times and corresponding deadlines, the scheduler can fit more in while squeezing in best-effort jobs as early and often as is safe. Most schedulers without job runtime information strictly prioritize production jobs and hope for the best, generally suffering more SLO (deadline) misses and much worse best-effort

performance.

Gang scheduling with back-filling. A common requirement of high-performance computing (HPC) jobs , such as large-scale physical simulations and parallel SGD-based optimizations, is that all tasks making up a job be initiated and executed simultaneously [48, 50]. Such *gang scheduling* requires a sufficient set of machines to be available to assign at once. To avoid starving large jobs, HPC schedulers often reserve machines until the required number become free. Since leaving machines idle is undesirable, HPC schedulers may employ *backfilling* [19, 43, 65, 83] to opportunistically execute small jobs on the otherwise-idle compute capacity that will be used for the large job. Intuitively, backfilling "fills in the gaps" created when the scheduler is collecting machines to initiate a large gang-scheduled job. Gang-scheduling and backfilling are most effective when job runtimes are known. This information allows the scheduler to know (a) how long the "gap" resources would be idle (i.e., how long until the full set of machines is ready for the large job) and (b) which pending "small" job(s) could complete within the gap resources. Failure to accurately match small jobs to gaps can lead to using them for jobs that do not finish in time, wasting opportunities to improve service quality for users even when checkpointing avoids loss of work.

Elastic sizing of jobs based on progress. Jobs that expose progress metrics and are elastically parallelizable can be dynamically resized to achieve completion-time targets [20]. The progress metrics enable prediction of how much longer an executing job has before completion, assuming that the current rate of progress continues. Coupled with knowledge of how adding additional machines would increase the rate of progress, such predictions can be used to make adaptive job sizing decisions. We differentiate this use of runtime predictions from our focus on those that are available when deciding when and where to start a job—that is, available before the job enters a Running state.

2.4.1 Predicting job runtimes

In some environments, especially HPC and grid computing environments, users have been expected to provide runtime information explicitly. Naturally, the quality of such user-provided information varies widely, and automated approaches to generating runtime predictions have become desired by industry practitioners and datacenter operators. Automation eliminates the possibility of users "gaming the system" and significantly increases runtime estimate accuracy. Erroneous– and especially malicious– user-supplied runtime predictions may cause high priority jobs to be delayed or denied allocation, as lower priority jobs under-represent their runtime. Preemption helps but wastes resources, as best-effort jobs are terminated and have to be restarted later from scratch. Rayon [10] took the approach of enforcing declared runtimes through a reservation system that guarantees capacity allocations for jobs via *a priori* resource time profiles. Accurate runtime estimates are still required, though, for proper operation.

Strategies for predicting job execution times may be categorized according to how much is assumed or known about a job. First, some techniques [10, 14, 34, 38, 75] are designed for explicitly repeating jobs, such as in a scripted simulation parameter sweep or regular post-processing of an output file. So, each such job is a *recurrence* of a nearly identical job with known historical runtime information. This category has been used in HPC and Grid computing [38, 75] as well as cloud computing [10, 14, 34].

Second, performance modeling based on *white-box techniques* assumes that the structure of each job is known. This information, together with input file characteristics, feeds performance models used for runtime prediction. For example, Jockey [20] and Perforator [18] leverage job structure and combine it with profiling for accurate runtime predictions. MapReduce's map-shuffle-reduce structure is well-understood and lends itself to analytical performance models, such as ARIA [76] and Parallax [47]. Similarly, Apollo [6] and Ernest [74] rely on leveraging job structure knowledge to estimate job runtimes.

A third category of runtime predictors uses *black-box techniques* to address the many jobs

that neither (a) report explicit recurrence nor (b) arrive with known white-box models. Few such predictors have been reported in the literature. Harchol-Balter and Downey [28] showed that, in the absence of other information, a reasonable approach to predicting a job's *remaining* runtime is to assume it is half-way completed. But, such an approach does not provide pre-execution runtime predictions (or particularly accurate predictions on a per-job basis). The closest prior work categorized HPC jobs, such as by user or by site, during post-processing of Grid traces, to characterize and determine the predictability of job runtimes and queue wait times [61]. We've prototyped a new predictor that differs from prior approaches by applying online learning techniques to identify adaptively the best job characteristics by which to categorize and the best predictor to use for each resulting category. This is a separate body of work. Preliminary results indicate that our predictor is capable of generating sufficiently accurate predictions even for the extremely diverse collection of jobs submitted to a Google production cluster [56].

2.4.2 Addressing runtime mis-predictions

Of course, no runtime prediction mechanism will be perfect in practice. Performance jitter in real systems can cause runtime variation, even for recurrences of the exact same computation. Greater variance is naturally expected as the breadth of jobs treated as a single category grows. At least occasionally, jobs will be submitted that do not fit into any existing category for which a predictor has sufficient prior information to make good predictions. And, of course, there are occasional major outliers... jobs that are expected to behave like recent similar jobs sometimes instead behave very differently, running much longer or shorter due to unexpected bugs, corner cases, user input, or other hidden changes.

Mis-predicted runtimes can lead to sub-optimal scheduling. For example, a job with an under-estimated runtime might not be started early enough to finish by its deadline. Job runtime over-estimates may result in other jobs being run less efficiently on non-preferred resources rather than waiting for preferred (but expected to be occupied due to over-estimates) resources. In general,

when a packing algorithm makes decisions based on inaccurate job "shapes", cluster scheduler efficiency may suffer significantly. Yet, we are aware of no work that analyzes and addresses specific effects of mispredictions on cluster scheduling.

Preemption is one standard tool that can be applied (and has been applied) to address some issues arising from mispredictions, either by killing (e.g., in container-based clusters [77]) or migrating (e.g., in VM-based systems [80]) jobs. Traditionally, preemption is used to re-assign resources occupied by lower-priority jobs to higher-priority jobs. However, it can also be used to address situations where under-predicted runtimes result in a job unable to complete within the time-window planned for it or to give preferred resources to jobs that would benefit more– thereby raising overall cluster efficiency. We introduce a new approach to making preemption decisions in §3.5 that regularly reviews the schedule holistically and determines the overall cost and benefit of potential preemption-scheduling combinations.

If any jobs are in danger of missing their SLO deadlines, the scheduler will determine which job(s) to preempt based on the values of all jobs involved. Based on this valuation, the scheduler may, for example, (a) terminate a single mis-predicted job, (b) terminate a number of small jobs, or (c) simply avoid scheduling new jobs. Our value-based approach enables the scheduler to approach the decision as a global optimization.

Preemption alone is insufficient, and we introduce several refinements (§5.2.5) specifically aimed at mitigating the negative effects of mispredictions. For example, when a job exceeds its (under-)predicted runtime, careful online re-prediction is needed to make good decisions regarding preemption. For jobs with short deadlines, some hedging is needed in the benefit model to increase the likelihood that the scheduler will dispatch those jobs despite long predicted runtimes, because those runtimes may be over-estimated.

Chapter 3

Design: Space-Time Request Language

3.1 Dynamic Partitioning

3.1.1 Equivalence sets

With the heterogeneity and dynamicity of shared resources on the rise [57, 59], scheduler designs can no longer assume that resources in the pool are interchangeable. Creation of work queues statically tied to certain logical cluster partitions breaks down as tens of machine attributes observed in large scale clusters today change over time. At best, the maintenance of these work queues becomes a burden.

From another angle, the heterogeneity of workloads and their placement requirements makes it clear that the machine attributes as well as the extent to which workloads care about them varies greatly across jobs and over time. Evidence from available trace analyses [56, 57] suggests that placement constraints can range from absent (maximum flexibility) to unachievable (zero flexibility), even in the empty cluster with no running jobs. In short, the logical partitioning of the cluster must be done from the perspective of the workload itself. It's a property of a job and a function of time.

To address this, we adapt the notion of *equivalence sets*. Each resource consumer can group

resources into equivalence sets, based on its particular concerns, such that the individual resource units within each equivalence set are fungible. In other words, the defining characteristic of an equivalence set is that any k-subset of the given equivalence set is indistinguishable and identical from the perspective of a job. Referring back to the discussion of combinatorial heterogeneity in §2.1, equivalence sets effective enable grouping multiple elements of the superset together and mapping them all to the same value. This reduction mechanism is significant, as exponentially many spatial placement options can be coalesced typically reducing the spatial complexity of soft constraint expressions. Thus, equivalence sets are the first mathematical primitive building block in the design of our space-time request language. Such resource grouping can range from per-machine sets on one extreme to having the entire cluster as a single set on the other. Given this mapping, all subsequent primitives are defined over a (likely much reduced) set of equivalence sets.

3.1.2 Partitioning Equivalence Sets

With multiple simultaneous resource requests, equivalence set specifications by multiple users intersect. In Fig. 2.1 an MPI job's fallback could be to run anywhere. The latter is an equivalence set comprised of all machines. The scheduler thus needs to partition a set of intersecting equivalence sets into a set of partitions p_i. Partitions have the following properties:

- there exists a subset of partitions p_j, such that $\cup_j p_j$ exactly equals that equivalence set;
- $\cup_i p_i = \cup_j eq_j$ for all submitted jobs j *currently* considered for placement. Note that we repartition the cluster for each scheduling decision in order to *minimize* the number of partitions, thereby reducing the complexity of the scheduling problem.
- Lastly, partitions are, by definition, non-intersecting.

The Scheduler consumes equivalence sets and dynamically repartitions them on demand. The resulting partitions are subsequently used to formulate the MILP problem and solve for the schedule. The role of partitions on MILP problem size is substantial, as all machines within a given

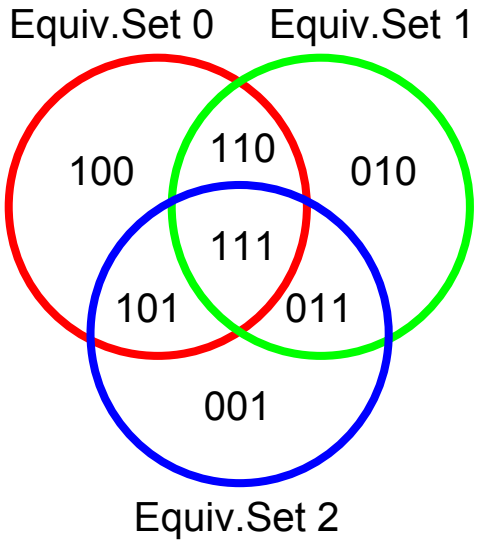


Figure 3.1: Individual circles represent equivalence sets that have overlapping machines. Each region is a partition and is uniquely determined by the bitvector of equivalence sets. Since each such region consists of machines that share the exact same set of attributes (same set of bits is turned on), we sometimes refer to partitions as *samesets* in this dissertation.

partition can be considered equivalent by the scheduler. We will be using this partition notation in our language specification to make it clear what gets translated to the MILP problem.

Thus, instead of statically defining partitions, we develop an algorithm to dynamically construct partitions based on the equivalence sets of currently pending jobs. This keeps the number of partitions to a minimum, reducing the burden on the scheduler core. Fig. 3.1 illustrates the relationship between equivalence sets and partitions with a Venn diagram.

Algorithm 1 converts a set of equivalence sets into a set of partitions. The key insight is that a partition is uniquely identified by the set of equivalence sets of which it is a member. Each bitvector position corresponds to a given equivalence set. To illustrate this, Fig. 3.1 labels each partition with a bitvector encoding the equivalence sets it is a member of. Our algorithm then creates these bitvectors and assigns partition numbers to each unique bitvector. When it completes, it returns a mapping of machine sets to unique partitions.

Algorithm 1: Partitioning algorithm

```
1  partition: equivClasses → partitions
2  func partition(equivClasses):
       // Create bitvectors for each machine corresponding to the equivalence
       // sets of which it's a member
3      bitvectors := new array of bitvector
4      foreach index, equivClass in equivClasses :
5          foreach machineId in equivClass :
6              bitvectors[machineId].setBit(index)
       // Machines with the same bitvector assigned same partition number
7      partitions := new array of int
8      hashtable := new hashtable from bitvector to int
9      nextPartition := 0
10     for machineId := 0 to cluster size :
11         partition, found := hashtable.find(bitvectors[machineId])
12         if ! found :
               // Bitvector hasn't been seen before;
               // assign new partition number to bitvector
13             partition = nextPartition
14             hashtable.insert(bitvectors[machineId], nextPartition)
15             nextPartition++
16         partitions[machineId] = partition
17     return partitions
```

3.2 Language Requirements, Goals, and Intuition

STRL's design is governed by the following five requirements that capture most practical workload placement preferences encountered in datacenters [59] and HPC clusters [55]:

[R1] space-time constraints

[R2] soft constraints (preference awareness)

[R3] combinatorial constraints

[R4] gang scheduling

[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 above in § 3.1.1), their quan-

tity, 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 [68]. 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.

3.3 Language Specification

3.3.1 Language Primitives

Resource consumers operating in heterogeneous resource context of any kind need the ability to specify utility over both the quantity and the type of resources allocated. The equivalence sets provide ability to *dynamically* and logically partition the cluster into "types", and we define primitives to map the quantity of resources chosen from a given equivalence set to utility. Then, a composition of primitives, defined with the operators introduced in subsection 3.3.2, aggregates the utility across potentially many equivalence sets.

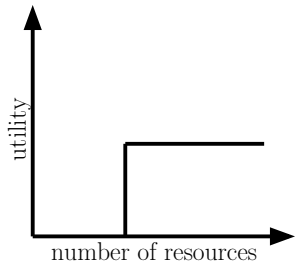


Figure 3.2: n Choose k primitive: utility function associating utility u with $\geq k$ resources allocated from the specified equivalence set.

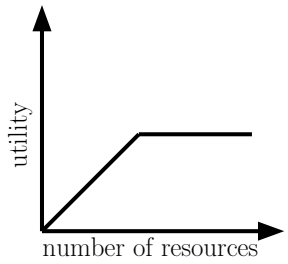


Figure 3.3: Linear "n Choose k" primitive: piece-wise linear utility function associating utility u with $\geq k$ resources allocated from the specified equivalence set, with a linear aggregation with $< k$.

Primitives form the leaves of the utility function expression. They provide the ability to express utility associated with the quantity of resources chosen from a specified equivalence set. For our preliminary evaluation, we have implemented two such primitives: the "n Choose k" (nCk) primitive and its linear counter-part. The nCk primitive maps an equivalence set and a number (k) of resources from that set to a utility value. Thus, given an assignment of resources across all equivalence sets, this primitive returns either 0, indicating that its request was unsatisfied, or the utility value, indicating that the provided assignment has issued $\geq k$ resources from the specified set. Pictorially, the utility function encoded by the nCk primitive is shown in Figure 3.2, while the linear "n Choose k" in Figure 3.3 allows for linear aggregation of resources from the specified equivalence set up to k, at which point it levels of at specified utility u.

We find that keeping the set of primitives as small as possible is advantageous for the orthogonality of the design. That said, we envision that other primitives may become more convenient

expressions of certain soft constraints in the future.

A more formal specification follows. A primitive STRL expression can be:

1. nCk(equivalence_set, k, $start$, dur, v): out of the specified equivalence_set, choose k machines starting at start time $start$ for duration dur to get value v. It is the main STRL primitive we call "n Choose k" used to represent a choice of any k resources out of the specified equivalence set, with the start time and estimated duration. nCk is useful for any job that needs or benefits from using machines with particular characteristics. In Fig. 2.1, 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. This serves as an example of how STRL captures *simple* space-time constraints [R1]. It helps to visualize the nCk building block as a function assigning scalar values to arbitrary rectangles in resource space-time. In Fig. 2.1, each of the (potentially non-contiguous) rectangles can be expressed with an nCk primitive (see §3.3.3).

2. $LnCk$(equivalence_set, k, $start$, dur, v): out of the specified equivalence_set, choose up to k machines starting at start time $start$ for duration dur to get value v. A choice of $k' < k$ returns value $\frac{v}{k} \cdot k'$. This "linear nCk" primitive is useful for further coalescing spatial placement options in terms of resource quantity drawn from one specified equivalence set, as long as there's a well-defined function mapping partial sub-k allocations to $\mathbb{R}$. It is useful to keep in mind that any LnCk primitive can be unrolled into a composite MAX expression (§3.3.2) over k nCk primitives. Therefore, strictly speaking, LnCk is a convenience primitive, reducing expression complexity by O(k) in cases where placement options are flexible in the number of resources desired.

3.3.2 Language Operators

While the primitives allow association of numerical utility with the quantity of resources chosen from a given equivalence set, the full expressiveness of STRL's specification mechanism

comes from the way they are composed. In fact, STRL language primitive are sufficient to express simple constraints, but not combinatorial constraints. Furthermore, specification of fallback placement options cannot be done with the "n choose k" primitive alone. Composition of these primitives enables operating on two or more subsets and, therefore, enables combinatorial and soft constraints.

To compose these primitives, we introduce operators with the following intuitive semantics: Min, Max, Sum, Scale, and Barrier. Each of these operators take numerical utility values as input, and outputs a single utility value. The first three can have an arbitrary number of operands. On input, they take a set of children that evaluate to a scalar utility value and perform the corresponding min, max, or sum operation over them. Scale and Barrier are unary operators. Scale multiplies the utility of the child by a specified scalar factor. Barrier() evaluates to zero until the utility of the child reaches a certain barrier, at which point it returns the specified utility on output.

A more formal specification follows. A composite STRL expression can be:

1. 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. 2.1, 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:
   $e1 = nCk(\{M1, M2\}, k = 2, start = 0, dur = 2, V)$
   $e2 = nCk(\{M1, M2, M3, M4\}, k = 2, start = 0, dur = 3, v)$
   and $V > v$. This example will be expanded in §3.3.3.

   General space-time elasticity of jobs can be expressed using MAX to select among possible 2D space-time shapes specified with nCk. Indeed, iterating over possible start times in the pictured range($start \in [0, 4]$) adds more placement options when we add the time dimension.

2. 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. 2.1, 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. This is an example of how STRL captures *combinatorial* placement constraints.

3. 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. This is used for advanced combinatorial constraint specification, such as allocation of "up to but no more than K' " borgmaster [77] servers in a given failure domain, with a total number of borgmaster servers totaling K. We're not aware of any other scheduling language that is capable of expressing such complexity.

4. 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. Combined with BARRIER, SCALE is used for advanced combinatorial constraints [R3], such as high availability service placement with specified tolerance threshold for correlated failure count [68].

5. 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.

3.3.3 Space-Time Request Language Examples

Simple constraints

Suppose a GPU job arrives to run on a 4-node cluster in Fig. 2.1. 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:

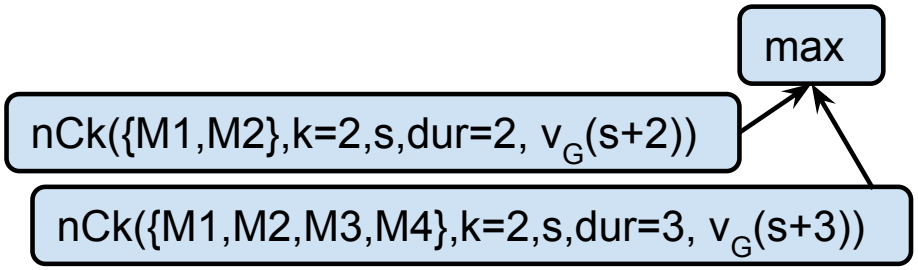
$nCk(\{M1, M2\}, k = 2, s, dur = 2, v_G(s + 2))$

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

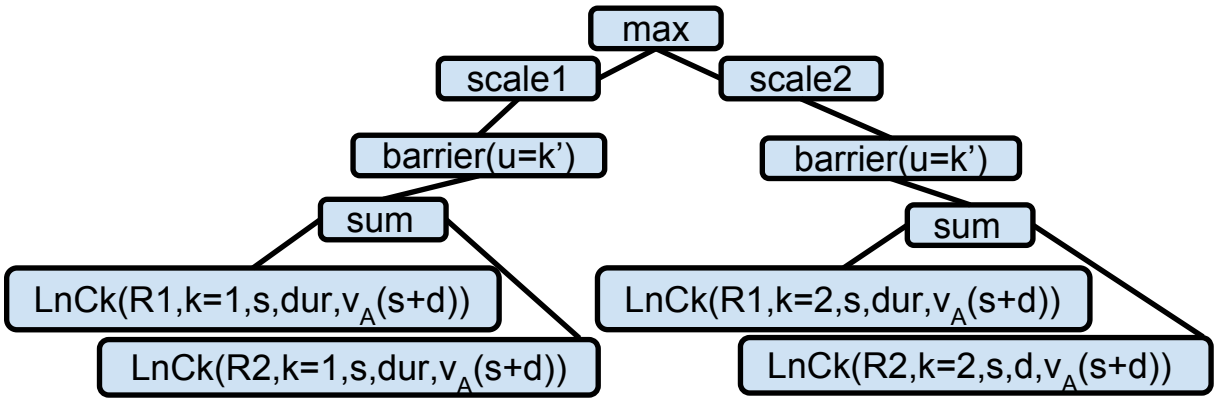
The first choice represents getting 2 GPU-enabled nodes, and completing in 2 time units with a start time s. The second choice captures all 2-combinations of nodes and represents running anywhere with a slower runtime of 3 time units. The STRL Generator combines these choices with a max operator, ensuring 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. 3.4(a), 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.

Combinatorial Constraints

Anti-affinity jobs that prefer their tasks to be distributed thinly across failure domains exemplify soft combinatorial constraints. A STRL expression for a more generalized version of the Availability job in Fig. 2.1 is shown in Fig. 3.4(b). The STRL expression for this type of job consists of a max expression with two branches corresponding to having up to $k = 1$ or $k = 2$ servers per rack. This limit is job-specific and is expected to depend on the maximum number of service instances the job can tolerate losing at any given point in time. To quantify the value of



(a) GPU



(b) Availability

Figure 3.4: STRL expression examples for jobs shown in Fig. 2.1 demonstrating simple and combinatorial soft constraints.

availability, users specify value degradation as a function of availability. This function is used to determine the scaling factors for each of the two main branches. Within each branch, we have a sum operator that sums up the number of servers aggregated across all racks (used as an example of a failure domain here). The barrier operator ensures that the total number of servers aggregated across all racks is at least the requested k'. The LnCk (linear "n Choose k") leaf nodes correspond to each rack and limit how many servers can come from any given rack to k. The translation from k' to job's budget is performed in the scale operator along with value attenuation due to respective availability degradation.

3.4 Deriving STRL from YARN Jobs

3.4.1 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 §3.3.3 (Fig. 3.4(a)) arrives with a deadline=3 time units (Fig. 2.1). Then, its AM-specified RDL [10] 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 $[s; f]$. As discussed in §2.4.1, estimates for different placement options can be learned by production

[1] MapReduce framework AM is a prime example of this.
[2] Please refer to [10] for complete RDL specification.

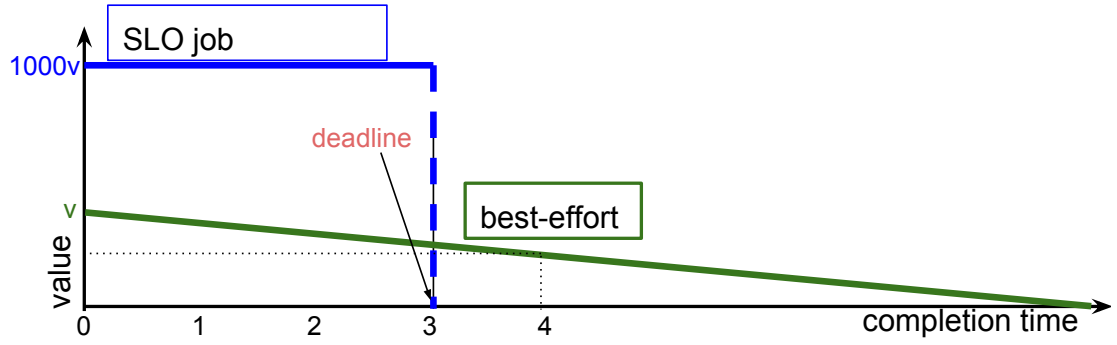
cluster systems (e.g., Perforator [18]) over time for recurring production jobs. For some jobs, analytical models show accurate results across varying input sizes [76], and a number of systems have implemented a combination of performance modeling and profiling [20, 60]. Runtimes for unknown applications can be inferred from slowdown factors induced by heterogeneity [15, 16] coupled with initial estimates learned from clustering similar jobs (work in progress). In our experimental TetriSched/YARN stack, information about estimated runtimes and deadlines comes from reservation requests submitted by the ApplicationMaster to Rayon [10].

3.5 STRL Preemption Support

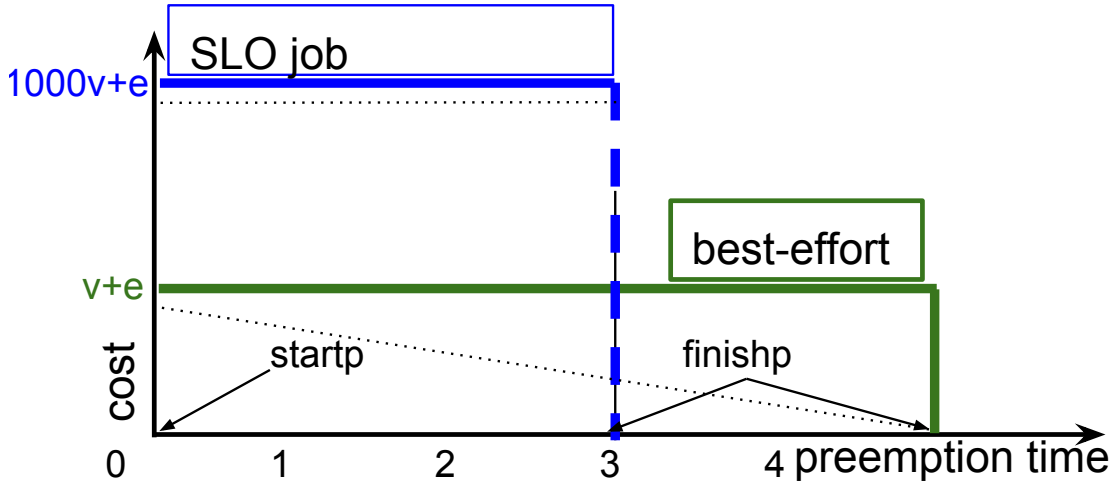
For TetriSched, we also need a language primitive that enables us to capture the cost associated with preempting a currently used space-time allocation. We introduce a new preemption primitive, KillnCk, that captures two-dimensional space-time preemption options. Its generality enables jobs to declaratively specify equivalence sets of resources and the quantity that can be preempted from that set. For example, suppose that a job currently holds GPU and non-GPU resources, where GPU nodes are used for tightly coupled non-elastic compute-intensive activity. The job can specify that preempting anything from the GPU equivalence set is very expensive, while preempting non-GPU resources is a lot cheaper. Thus, KillnCk can encode $O(\binom{n}{k})$ possible k-sized preemption options from an equivalence set of n nodes with as few as 1-2 KillnCk primitives.

3.5.1 Preemption cost functions

We must inform the scheduler about the cost associated with preempting a given job. We model the cost of preemption as a function of time $c(t)$ (Fig. 3.5). For a given job, $c(t)$ is defined only in the range specified by the KillnCk interval $[startp; finishp]$. The job can only be preempted within this interval of time. SLO jobs are modeled to have the highest cost of preemption,



(a) value functions



(b) preemption cost functions

Figure 3.5: Internal TetriSched preemption cost functions for SLO and BE jobs.

while best-effort jobs—the lowest. **The initial value of $c(t)$:** is empirically chosen to impose strict initial priority among jobs. For example, the initial value for SLO jobs admitted by the admission control system is set to 1000$\times$ the initial value of best effort jobs. We add a small ϵ to ensure that jobs already running cannot be preempted by equally valued pending jobs.

3.5.2 Preemption STRL Primitives

Formally, TetriSched preemption STRL primitives include:

1. A "kill n Choose k" expression of the form:

   $KillnCk$(equivalence_set, k, $startp$, $finishp$, c). It is the main in situ preemption prim-

itive used to represent a choice of any k resources out of the specified equivalence set for preemption, where $startp$ marks the earliest possible time when preemption is allowed, and $finishp$ marks the last opportunity to preempt this job – typically its expected completion time.

2. A linear "kill n Choose k" expression of the form:

   $LinearKnCk$(equivalence_set, k, $startp$, $finishp$, c). It is a variant useful for encoding willingness to preempt up to but not exceeding k nodes from an n-sized equivalence set of nodes. The cost of preemption c is sized linearly with the number of nodes chosen by the solver to be preempted. This preemption STRL primitive is ideal for elastic jobs that can make progress with a smaller footprint, such as MapReduce jobs.

Chapter 4

Mixed Integer Linear Programming Formulation

A canonical MILP instance consists of three main components: decision variables—the unknowns the solver attempts to provide an answer for, constraints—linear inequalities based on decision variables that must be satisfied by the solver, and an objective function—a linear function on decision variables optimized by the solver. The tree structure of STRL algebraic expressions lends itself nicely to a recursive MILP generation algorithm. Starting from the root of the expression, we gradually (a) create new decision variables as needed for each node of the tree, (b) construct various types of constraints (e.g., to enforce the semantics of STRL operators), and (c) recursively construct an objective function doing a depth first traversal of the STRL tree.

We associate a binary decision variable I with each branch of the tree to allow the solver to cut or include entire subtrees. Setting an indicator variable to 1 for one of the MAX operator's children, for instance, effectively chooses a placement option encoded by that child. To capture the quantity of resource allocation from individual equivalence sets, we use partition decision variables. A given partition variable P_p^t encodes the amount of resource a solver will allocate from partition p at time t for duration dur of the nCk primitive processed. Constraints then build on indicator variables I and partition variables P_p^t to construct two main types of constraints. *De-*

mand constraints ensure that the sum of P_p^t over all partitions p that make up a given equivalence set equals to the requested k nodes. *Supply* constraints ensure that the sum of all partition variables P_p^t do not exceed capacity $cap(p,t)$ of partition p at time t. Lastly, the objective function for nCk is the value associated with the placement option it encodes, multiplied by the indicator variable I that governs the flow of value up from this primitive. It helps to visualize the objective function as the flow of value from its primitive leafs up the tree and aggregated at the root. On its way up, the flow is modified by various operators that channel the maximum flow, minimum flow, or aggregate flow from its children.

4.1 Automatic MILP Generation

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 2). Recall from §3.3 that STRL expressions

are expression trees composed of STRL operators and leaf primitives. 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,[1] 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 (discussed in §3.1.1) 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

[1]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.

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.

4.2 MILP Generation Example

Suppose 3 jobs arrive to run on a cluster with 3 machines {M1, M2, M3} (Fig. 4.1):

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

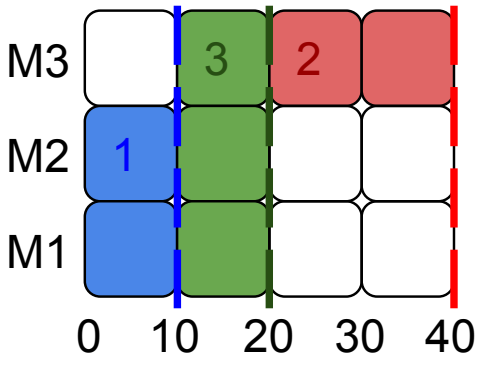


Figure 4.1: Requested job shapes, deadlines, and final order.

Algorithm 2: MILP generation algorithm

```
gen: (STRL expression tree, indicator var) → objective function
func gen(expr, I):
    switch expr:
        case nCk(partitions, k, v, start, dur)
            foreach x in partitions:
                P_x := integer variable  // Create partition variable per partition in equiv set
                for t := start to start + dur:
                    Add P_x to used(x, t)  // Add partition variable to used to track supply constraints
            Add constraint Σ_x P_x = k * I  // (Demand) Ensure this node gets k nodes if chosen
            return v * I  // Return value if chosen, i.e., I=1
        case LnCk(partitions, k, v, start, dur)
            foreach x in partitions:
                P_x := integer variable  // Create a partition variable per partition in equiv set
                for t := start to start + dur:
                    Add P_x to used(x, t)  // Add partition variable to used to track supply constraints
            Add constraint Σ_x P_x ≤ k * I  // (Demand) Ensure this node gets up to k nodes
            return v * Σ_x P_x/k  // Return scaled value based on the number of allocated nodes
        case sum(e_1, ..., e_n)
            for i := 1 to n:
                I_i := binary variable  // Create indicator variable for each branch
                f_i = gen(e_i, I_i)
            Add constraint Σ_i I_i ≤ n * I  // Ensure that no branches are chosen if I = 0
            return Σ_i f_i
        case max(e_1, ..., e_n)
            for i := 1 to n:
                I_i := binary variable  // Create indicator variable for each branch
                f_i = gen(e_i, I_i)
            Add constraint Σ_i I_i ≤ I  // Ensure that at most one branch is chosen
            return Σ_i f_i
        case min(e_1, ..., e_n)
            V := continuous variable  // Create variable representing min value
            for i := 1 to n:
                f_i = gen(e_i, I)  // Choose branches based on the same indicator variable I
                Add constraint V ≤ f_i  // Ensure V is less than the min value
            return V  // Given the constraints, maximizing V makes V equal to the min value
        case scale(e, s)
            return s * gen(e, I)  // Scale the objective function by s
        case barrier(e, v)
            f = gen(e, I)
            Add constraint v * I ≤ f  // Ensure the child expression meets barrier constraint
            return v * I  // Barrier caps the value at v
I := binary variable  // Dummy indicator variable
f = gen(expr, I)
foreach x in partitions:
    for t := now to now + horizon:
        // (Supply) Add supply constraints to ensure usage ≤ avail resources
        Add constraint Σ_{P∈used(x,t)} P ≤ avail(x, t)
solve(f, constraints)
```

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 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 2). 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. 4.1) of running job 1 immediately, running job 3 at time 10, and running job 2 at time 20.

4.3 Preemption Support

In Algorithm 3 we describe the MILP generation algorithm for KillnCk— the main preemption primitive that declaratively captures the cost associated with preempting a currently running job. Intuitively, the indicator variable I controls the contribution of a given KillnCk primitive to supply constraints and the overall objective function. As expected, the objective function term contributed is negative and reflects preemption cost. The partition variable P_x captures the amount of resource that will be preempted from the specified equivalence set. The equivalence set itself is constructed from the set of nodes currently occupied by the node. Thus, the supply constraint effectively contributes a negative amount of resource capacity to the corresponding vertical slice of resource space-time, if chosen. This has the desired effect of returning resources to the available pool, if the indicator variable I was chosen by the solver to equal to 1. Lastly, the demand constraint enforces the all-or-nothing semantics of preemption. We make sure that the number of tasks preempted is exactly the number of tasks in this job.

Finally, the Mixed Integer Linear Program portion contributed by KillnCk is recursively combined with MILP contributions from other parts of the aggregate STRL expression. It results in a single canonical MILP instance that includes contributions from all preemptible jobs as well as all pending jobs. The result of the MILP solver is a simultaneous determination of how much to preempt from each of the running jobs and which pending jobs to schedule instead—all done in

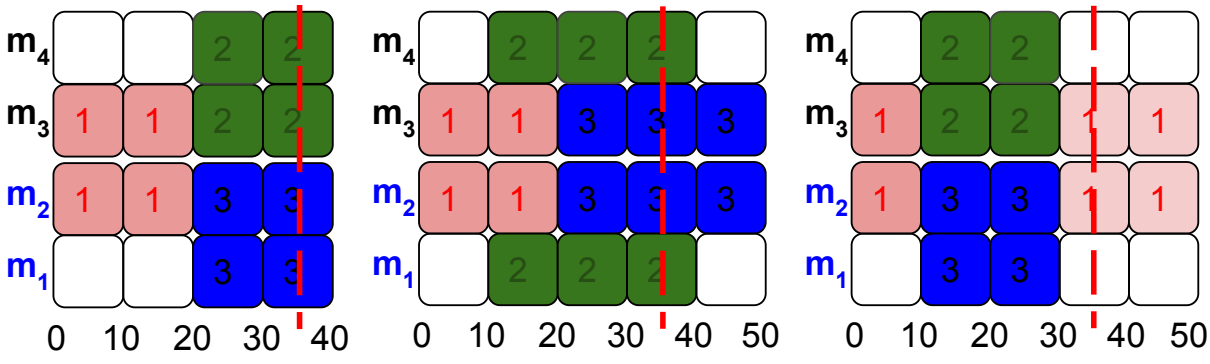
Algorithm 3: MILP generation: preemption primitive

```
1  gen: (expr , indicator var) → objective function
2  func gen(expr, I) :
3      switch expr :
4          case KillnCk(partitions, k, startp, finishp, c)
5              foreach x in partitions :
6                  P_x := integer variable  // Create partition variable
7                  for t := startp to finishp :
                       // (Supply) Track usage
8                      Add −1 ∗ P_x to used(x, t)
               // (Demand) Ensure this node
               // gets k servers if chosen
9              Add constraint Σ_x P_x = k ∗ I
10             return −1 ∗ c ∗ I  // Return value if chosen
   // Main function
11 I := binary variable  // Create indicator variable
12 f = gen(expr, I)
13 foreach x in partitions :
14     for t := now to now + horizon :
           // (Supply) Ensure usage ≤ avail resources
15         Add constraint Σ_{P∈used(x,t)} P ≤ avail(x, t)
16 solve(f, constraints)
```

a single scheduler cycle.

4.4 Preemption MILP Generation Example

This section illustrates TetriSched's simultaneous cost/benefit consideration of preemption and placement with a simple example. Consider a small heterogeneous cluster and these 3 jobs: job 1, a best effort job, arrives and starts running at time 0. Job 2, an MPI job, can run in 20s if scheduled on the same rack or in 30s otherwise. Job 3, a GPU job, can run in 20s if scheduled on GPU nodes or in 30s otherwise. Jobs 2 and 3 are SLO jobs arriving at t=10 with a deadline at t=35. There are three options, all shown in Fig. 4.2. First, SLO jobs 2,3 may wait for their respective preferred resources to become available (Fig. 4.2(a)). Second, they may be scheduled on available resources as soon as possible (Fig. 4.2(b)), taking a performance hit



(a) no preempt, hard constraint (b) no preempt, soft constraint (c) with preemption

Figure 4.2: Three jobs on a small heterogeneous cluster: 2 GPU machines (m1 and m2) on rack 1 and 2 non-GPU machines on rack 2. Job 1 (pink) arrives at t=0 and has no deadline (best effort). Job 2 (green) and Job 3 (blue) arrive at t=10 and have a deadline of t=35, indicated by the vertical red line; they also run faster if using preferred resources (both tasks on same rack for job 2, both tasks on GPU machines for job 3). TetriSched is able to meet all deadlines only with preemption. If jobs 2,3 wait for preferred resources or run on suboptimal allocation, they fail to meet their deadline. Deadline can be met only if job 1 is preempted.

and missing their deadline. Third, the scheduler may preempt the half-done best-effort job and simultaneously place SLO jobs 2,3 (Fig. 4.2(c)). This third option is the only way their deadlines can be met.

It's worth noting that the probability of the best-effort job to be scheduled "inconveniently" is surprisingly high. The probability of the BE job breaking rack locality is

$$P(\text{BE job breaks rack locality}) = 1 - \frac{2}{\binom{4}{2}} = \frac{2}{3}$$

At the time when jobs 2,3 arrive at t=10, the resulting STRL expressions will be as follows.

- Running job1: $e_1 =$

  KillnCk({m2, m3}, k=2,startp=10,finishp=10,c=11)

- Pending MPI job2: $e_2 =$

max(nCk({m1, m2}, k=2,start=10,dur=20,v=100),

nCk({m3, m4}, k=2,start=10,dur=20,v=100),

nCk({m1,m2,m3,m4}, k=2,start=20,dur=20,v=1))

- Pending GPU job3: $e_3 =$
  max(nCk({m1, m2}, k=2,start=10,dur=20,v=100),

  nCk({m1,m2,m3,m4}, k=2,start=20,dur=20,v=1))

Thus, the resulting STRL expression to be maximized will be: sum(e_1, e_2, e_3). The MILP Compiler will then call the recursive MILP translation function gen(sum(e_1, e_2, e_3)) (Alg. 3, line 2). As the algorithm recurses down the expression tree to its leaves, it will process the killnCk primitive (line 4). Finally, the solver will determine that the highest possible value for this expression is $100 + 100 - 11$, where the negative term is contributed by the KillNck return call (line 10 in Alg. 3) and reflects the cost of preempting the running best-effort job. Upon completion of jobs 2,3 at t=30, the best-effort job is rescheduled again.

Chapter 5

Architecture and Implementation

To achieve our design goals and instantiate our theoretical building blocks in a real system, we've built the TetriSched core and integrated it with a popular open-source resource management framework, called YARN. YARN is widely deployed, including at Microsoft, and commercially supported by Cloudera and Hortonworks. Hadoop YARN has a thriving open-source community, with very active development in trunk.

This chapter describes the architecture, key components, and key enabling features of TetriSched. Our system artifact works in tandem with YARN's default reservation system [10] responsible for admission control. TetriSched 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 [10] 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.

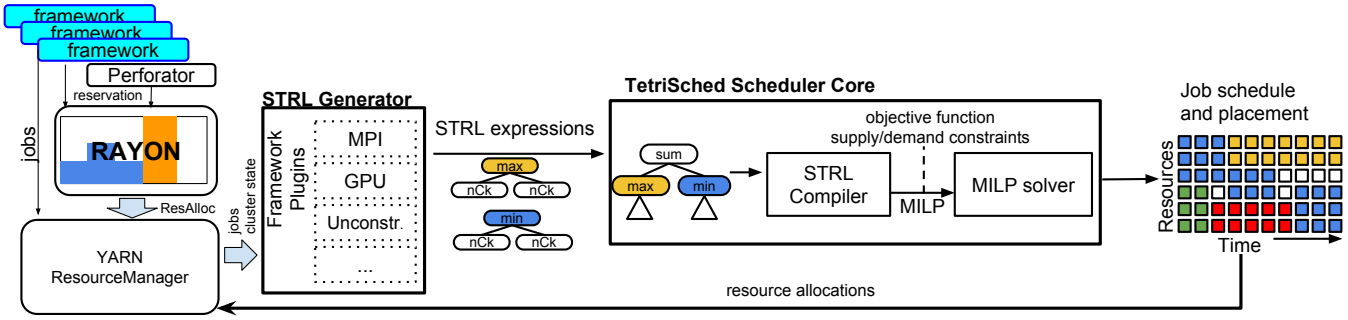


Figure 5.1: TetriSched system architecture

5.1 End-to-end system architecture

Fig. 5.1 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. 5.1 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. 5.3). 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. 5.3.

5.2 Scheduler Core

The scheduler core is responsible for: STRL aggregation, STRL to MILP compilation, MILP optimization, result extraction, and various optimizations to expedite the core scheduling cycle.

TetriSched core is a standalone scheduling server for algebraic scheduling that allocates resources in accordance with STRL expressions submitted as resource requests. The scheduler core manages its view of cluster resources and their attributes. This enables YARN's ResourceManager to compile appropriate equivalence sets for each resource request based on the resource attributes of interest to a particular ApplicationMaster asking for an allocation. Thus, scheduler core clients specify their resource requests in the form of algebraic STRL expressions—the only interface the scheduler core understands by design. Submission of these STRL expressions can occur at the following interesting points in time:

1. when the job enters the pending state, requesting resources
2. when the job enters a different stage of execution, and a set of preferred resources changes
3. when the resource consumer to relinquish resources to save on the cost of their allocation
4. when additional resources are needed

The scheduler core fires the allocation algorithm either at regular intervals or in the event-driven fashion. In either case, it will only execute when there is at least a single pending job OR an updated STRL expression to be scheduled. Having thus accumulated a non-empty set of utility functions, the scheduler can either globally or greedily perform the assignment of resources in a way that (globally or greedily) attempts to maximize the overall value of cluster resources. The return result of such an assignment is the resource allocation matrix, such that its $D\,[i, j]$ component represents the number of schedulable units allocated from equivalence set j

to scheduler core client i.

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 (§3.3.2) and assigns cluster resources so as to maximize the aggregated STRL expression value extracted from shared cluster resources (see §3.2 for intuition and detail). Value functions are a general mechanism that can be used in a variety of ways [68, 70], 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, Fig. 6.1), 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 (§4) 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. 7.4.3.

5.2.1 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 [66] or always waiting [81]). 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. 2.1) 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.

5.2.2 Global scheduling

The declarative algebraic nature of our STRL language combined with automatically compiling the resulting aggregate expression to a *single* MILP problem instance enables a unique feature of TetriSched to simultaneously consider many job's constraints at the same time. We refer to this feature as *global* scheduling. As we'll see in the evaluation chapter, the ability to side-step the inherent ordering of ANY greedy scheduling solution and, instead, reduce ALL pending requests to a single mathematical optimization problem empirically produces significant advantages, particularly for heterogeneous workloads.

Here, I develop some mathematical intuition for why scheduling with constraints is hard. Scheduling a job can itself be viewed as an algebraic operation that mutates cluster space-time. It can be proven under simplifying assumptions that scheduling unconstrained jobs with known runtime estimates and resource quantity demands is (a) commutative and (b) associative. Indeed, considering unconstrained jobs A, B, and C in any order for placement yields the same result in terms of cluster space-time allocation. Any order will yield the same total footprint shape (how many resources for how long). Scheduling *constrained* jobs, however, loses this desirable property. With 4 machines (two with a GPU) and two jobs A and B asking for 2 machines each (B preferring GPU), the order of scheduling them matters. Indeed, scheduling the unconstrained job A has a high probability ($= \frac{5}{6}$) of occupying at least 1 GPU node, causing job B to wait. Scheduling B first followed by A results in a better schedule, higher instantaneous utilization,

and both jobs completing simultaneously — a different space-time footprint shape. Clearly, even the simplest constraints in just a subset of jobs [56, 57] cause the scheduling operator to lose its commutativity. Placement constraints thus necessitate careful consideration of all known preferences simultaneously in order to achieve the best space-time shape possible. This is a new challenge for datacenters that requires new solutions that transcend shuffling job orders or quantity allocation within the same envelope of aggregate space-time load on the cluster.

It is worth noting that, contrary to popular belief, even if all job order permutations are considered, it is still possible, for ANY order, to make arbitrarily bad decisions for job J_i that negatively affect subsequent jobs J_{i+1}, J_{i+2}, ..., J_n. Only simultaneous consideration of all pending jobs ensures their optimal constraint resolution. This simultaneity proves to be particularly helpful when we consider preemption.

5.2.3 In Situ Preemption

TetriSched leverages its combination of declarative STRL resource requests and MILP formulation to support what we call "in situ preemption". TetriSched's in situ preemption leverages the same STRL building blocks and naturally plugs in to the cyclical aggregation of STRL expressions and global consideration of pending jobs for placement. Preemption is enabled on a per-job basis. When it's enabled, each running job is represented as a STRL expression using one of the two preemption primitives defined in §3.5.2. This STRL primitive, if chosen, contributes negative value to the aggregate STRL expression (thus making it costly to preempt), while simultaneously contributing negative demand on resource capacity for the occupied set of resources through the specified preemption interval (§4.3). As the overall STRL expression is optimized, the qualitative outcome of mixing preemptible jobs together with pending jobs is simultaneous consideration of running jobs for preemption and pending jobs for placement. As a result, the solver will determine the best candidates to preempt, if necessary, in order to accommodate the best candidates for placement.

Furthermore, A simple extension to our MILP compilation algorithm for preemption makes it possible to simultaneously re-consider a preempted job for re-placement in the same cycle as it's being preempted. This is desirable particularly in scenarios when there are deadline jobs that have loose deadlines, can be rescheduled, and still finish in time, while accommodating a more urgent SLO job. In the absence of this feature, the urgent job could be dropped (e..g, by the reservation system) or, if admitted to the best-effort queue, it could miss its deadline due to lack of resources for execution.

5.2.4 Greedy scheduling with preemption

TetriSched's preemption mechanism enables simultaneous consideration of pending jobs for allocation and running jobs for preemption. We leverage this simultaneity in our implementation of TetriSched's greedy mode. In greedy mode, TetriSched maintains two priority queues, one for SLO jobs and one for best effort. On each scheduling cycle, TetriSched selects the next SLO job in FCFS order and combines its STRL request with preemption STRL expressions for preemptible running jobs. The resulting expression's value is then maximized through the MILP solver. This achieves the desired effect of prioritizing SLO jobs for placement with simultaneous consideration of all preemptible running jobs for preemption. Given that the BE jobs' cost functions are significantly less than the value of an SLO job expected to complete by the deadline, this typically results in BE job preemption in favor of pending SLO jobs during periods of transient overloads or difficult to satisfy constraints (i.e., *effective* transient overloads).

Furthermore, a continuum of resource request aggregation is possible, starting from the more typical single job at a time greedy scheduling on one end of the spectrum to the full global scheduling mode (TetriSched's default)) that considers all pending jobs at once. This feature offers a tuning knob that unlocks the tradeoff between scheduling quality and scheduling latency, useful in environments with short jobs, where scheduling latency is more critical.

5.2.5 Handling Runtime Mis-Predictions

Under-Estimates

Under-estimates can cause significant SLO violations in time-aware schedulers that depend on estimate accuracy. Once the scheduler detects that an under-estimate occurs, it has a choice to kill it [10] or to optimistically let it complete [70]. The latter was shown to significantly improve SLO attainment, but we have found that it must be kept in check because there can be large under-estimates. On every cycle, TetriSched performs the cost/benefit analysis to determine whether an under-estimated job should be allowed to continue. Given a near-perfect estimate, the best choice is to increase a job's expected runtime minimally and let it finish. For large under-estimates, however, it is ideal to discover and react quickly. We, therefore, implement an exponential backoff policy that increments the expected runtime $\hat{T}$ by 2^t cycles, starting with $t = 0$ incremented on each cycle. This achieves the desired effect of hysteresis in the system. TetriSched reacts to minor under-estimates with minor runtime estimate corrections. As it learns that the under-estimate is more significant, it updates the runtime by progressively longer increments.

The effect of such exponentially longer increments is three-fold. First, it increases the window of opportunity for the scheduler to preempt this job, effectively growing $finishp$ in Alg. 3. Second, given monotonically decreasing value functions for SLO jobs that reach zero past deadline, larger runtime estimate increments will eventually surpass the deadline, either triggering a kill event or increasing the probability of preemption by other jobs. Third, increased estimates cost more space-time in the scheduler's plan-ahead window. Running jobs always create an opportunity cost, which pending jobs may outweigh. Increased estimated space-time increases the probability that there exists a set of smaller jobs that may occupy the same amount of space-time and offer higher value for it. The combined effect is progressively increasing the likelihood that an under-estimated job will be either killed or preempted, achieving the desired effect of pruning vastly under-estimated jobs that may inappropriately consume valuable resources. In other words, TetriSched evaluates the net benefit of preemption of the remaining *estimated* space-time

area of the late job. The key insight is that this remaining estimated space-time has a direct bearing on the scheduler's decision to preempt. Indeed, if a job is nearing completion, the benefit of preempting this job is far less significant, as the amount of space-time that frees is small relative to jobs with a much bigger estimate of remaining time to completion.

Over-Estimates

In §5.2.5 above, we mention the possibility of SLO jobs' termination when their $\hat{T}$ is expected to exceed the deadline. To avoid unnecessary termination of jobs due to over-estimates, including over-estimates arising from adjusted under-estimates, TetriSched changes the job's value function to have a linearly decaying slope past the deadline, instead of the sharp drop to zero. The resulting effect is that over-estimated jobs may continue to run while their *expected* completion time exceeds the deadline. As the actual runtime nears the deadline, the job's preemption cost decreases, increasing the probability of preemption in favor of higher valued jobs. This naturally creates the desired effect of reducing resources spent on jobs that are increasingly expected to miss their deadline. With higher probability of the latter, the job's diminishing cost of preemption increases the probability of preemption and resource reclamation for higher valued jobs.

5.2.6 MILP Solver

The internal MILP model can be translated to any MILP backend. We use IBM CPLEX in our current system prototype. Given the complexity and size of MILP models generated from STRL expressions (hundreds of thousands of decision variables on a 1000 node simulated cluster with hundreds of jobs), the solver is configured to return "good enough" solutions within 10% of the optimal after a certain parameterizable period of time (set to 4s in our real cluster experiments). 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. As the plan-ahead window shifts forward in time with each cycle, the space-time schedule from the previous

step becomes a feasible initial solution to prime the MILP solver. We find this optimization to be quite effective.

5.3 YARN Integration

For ease of experimentation and adoption, we integrate TetriSched with the widely popular, active, open source YARN [71] framework. YARN uses containers to represent resource bundles. Containers are used to encapsulate tasks of a job, which typically consists of multiple such tasks. Examples include MapReduce map tasks, MPI tasks, Apache web server instances, etc. We add a Proxy Scheduler that interfaces with the TetriSched daemon via a narrow Apache Thrift RPC interface. The interface is responsible for

1. Adding jobs to the TetriSched's pending queue with information about their type, deadline, estimated runtime, and the resource request.
2. Communicating allocation decisions to YARN's ResourceManager. YARN's RM stores TetriSched decisions in a lookup table and matches up heartbeating NodeManagers with the jobs they were allocated to serve.
3. Signaling job completion to TetriSched. This information is reported by the NodeManagers to the ResourceManager on a heartbeat. The RM then triggers an RPC call to the Scheduler to update its internal cluster state.

TetriSched makes allocation decisions based on thus provided information and its own view of cluster node availability it maintains.

Chapter 6

Experimental Setup

6.1 Simulation

We performed extensive simulation studies to evaluate TetriSched's schedules and those of competing approaches to handling placement constraints.

6.1.1 Cluster Configuration

Simulation allows us to study scheduling at much larger scale than we otherwise could. The results reported are all for a simulated cluster of $N = 1000$ servers spread across 25 racks: 10 racks each with 25 GPU-enabled servers, 5 racks with 40 servers, 5 racks with 60 servers, and 5 racks with 50 servers running an HDFS store. So, 25% of the cluster has GPU-accelerators, 25% of the cluster has HDFS local storage, and 50% of the cluster is generic, spanning 10 racks. Throughout our experiments, we keep this cluster composition constant and vary the workload composition to study the effect of *spacial* and *temporal* imbalances induced.

6.1.2 Simulated Workload Types

In simulation, we experiment with four prototypical workload types: Unconstrained, GPU, MPI, HA (high availability), and HDFS.

Unconstrained: Unconstrained is the most primitive type of job that has no placement 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 class.

GPU: GPU preference is an example of a simple soft constraint. In addition to the SLO parameters above, users specify that they want k GPU-enabled servers. If the k servers are not all GPU-enabled, then we assume the job is slowed down by a slowdown factor. This translates into a STRL MAX expression with two branches corresponding to k GPU-enabled servers and k servers anywhere (see Fig. 3.4(a)). Recall that the max expression carries the semantics of an "OR" operator. This pattern is repeated for each possible start time within the configured scheduling horizon window. The utility is calculated based on the start time, estimated duration, and user sensitivity to delay. TetriSched is responsible for evaluating these options in space-time to determine the best way to schedule pending jobs. Note that "GPU" here is representative of any arbitrary accelerator or machine attribute, such that performance benefit is achieved iff all k allocated machines have it.

MPI: Rack locality is a prime example of a combinatorial constraint. For instance, MPI workloads are known to run faster when their communicating tasks are co-located within the same locality (latency) domain. Here, users request k servers on the same rack and slow down if their allocated servers are on different racks. This translates into a STRL MAX expression with a branch per rack plus an additional branch for a placement anywhere in the cluster. Note that a "rack" could refer to any statically or dynamically determined locality domain.

HA (High Availability): Rack anti-locality is another contrasting example of a combinatorial constraint. Workloads that care about distributing servers across failure domains benefit

from this type of constraint. We describe the STRL expression for it in Fig. 3.4(b) and its accompanying text in §3.3.3. HA jobs are noteworthy as they combine two objectives: queueing delay and availability.

HDFS: The last example explores flexibility in both the number of servers requested and the type of resources consumed. In this example, jobs are able to consume fewer servers at the cost of running longer. Similarly to GPU jobs, HDFS jobs prefer to run on HDFS storage nodes where there is data locality. However, these jobs are able to extract partial benefit if some, but not all, of the tasks are on HDFS nodes. The STRL expression for this job consists of a max expression across a collection of HDFS/non-HDFS combinations. Each of these combinations is handled by a min expression, carrying the semantics of an "AND" operator. Intuitively, we perform a selection of the maximum-value pair (p, q), where p is the number of HDFS storage nodes and q is the number of non-HDFS nodes.

6.1.3 Workload Configuration

We now describe the pertinent parameters that affect our simulated cluster workload.

Definition of ρ: An important parameter that affects the impact of scheduling is load (ρ). In our simulation experiments, we adjust the job arrival rate (λ) to match a desired load (ρ). Formally, ρ is defined as

$$load = \rho = \frac{\lambda E[W]}{N}$$

where $E[W]$ is the average work per job and N is the cluster size. The work per job ($W = S * K$) is defined as the size of the space-time rectangle consumed by executing the job, which is the job duration (S) multiplied by the number of servers requested (K). Since a job may be flexible in S and K based on what resources it consumed and how many it consumed, we take S and K to refer to the optimal placement as indicated by the hard constraint. This implies that a non-optimal placement may increase the effective load on the system, if the job is slowed down. We define **slowdown** in Table 6.2.

Workload Mix	GPU	HDFS	HA	Unconstrained
W1	25%	25%	0%	50%
W2	25%	25%	50%	0%
W3	50%	0%	50%	0%
W4	100%	0%	0%	0%

Table 6.1: Workload compositions used in results section.

Workload Composition: Our simulated workloads are often composed of a heterogeneous mixture of job types. We experiment with many different proportions of workload types, as well as a broad range of settings of the other parameters. Table 6.1 shows the compositions used in our results section. Going from workload mix W1 to W4, the relationship of workload composition to resource composition becomes increasingly less balanced. For W1, at full load, we expect all three present job types to fit within their preferred spacial partitions, leaving no spatial imbalance. As we'll see in Sec. 7.1, TetriSched still outperforms alternatives through better handling of temporal imbalances caused by uncorrelated bursts in each workload type. W2 fits GPU and HDFS jobs to preferred resources, if HA jobs can fit on generic racks. This can happen when HA job sizes do not require spreading over more than the generic racks or if the scheduler exploits HA jobs' spatial flexibility to put more of the tasks on each generic rack. W3 is designed such that both GPU and HDFS jobs spill over to non-preferred resources at loads exceeding $\rho = 0.5$. Lastly, W4 is the least spatially balanced of all.

Job parameters: Workloads can also vary in their budgets, penalties, and deadlines. Unless otherwise stated, we set the budget as the job duration multiplied by the number of servers requested, which corresponds to the space-time rectangle consumed by the job. We fix the penalty to be equal to the budget [20]. We set the desired completion time to be the job duration, if the job runs on optimal resources. This indicates to the scheduler that we would like our results ASAP. We set the deadline to be two times the job duration, if the job runs on non-optimal resources plus the desired completion time. Thus, it is possible to extract positive value from jobs even if they are running on non-optimal resources, assuming they are quickly scheduled.

High availability jobs are unique in that they care about availability as well as queueing delay. We set the parameters so that the job would get full value if it runs with up to one server per rack. Under the flexible placement policy, a job is able to sacrifice availability to run with up to two servers per rack, but it suffers a loss in utility as a result. This availability vs utility tradeoff is configurable by the user. In our experiments, we configured the parameters so that running on up to two servers per rack yields a 10% loss in utility. We set the desired queueing delay to prefer starting ASAP, but these two parameters can be tuned by the user to prefer increased availability or reduced queueing delay.

Traces: We generate traces based on load, workload composition, job type, and burstiness. The trace file for each contributing workload type is generated independently. The traces include arrival time, estimated job duration, and the number of servers requested. Arrival times are generated based on each workload type's load and an **inter-arrival squared coefficient of variation parameter** (C_A^2), which controls the burstiness of the arrival sequence (Table 6.2). Setting $C_A^2 = 1$ yields a Poisson process, and higher values of C_A^2 yield burstier arrivals. To target an interesting range of job durations, we use a shifted exponential distribution with a minimum of five times the scheduling period and a mean of ten times the scheduling period. The number of servers per job varies based on job type and is bounded so that any given resource request, in isolation, can be satisfied by any one of the compared scheduling policies. Of course, real users may be unaware of the system configuration and may set a hard constraint that can't be satisfied even in isolation, but such ill-behaved jobs would penalize the hard constraint policy more than the others.

6.1.4 Availability Calculation

To evaluate consideration of placement tradeoffs across multiple failure domains (e.g., racks) for availability-sensitive jobs, we need a mechanism to quantify the relative impact of correlated failures. To do this, we developed a Monte Carlo-based simulation to estimate the availability

E[T]	Mean response time (completion time − arrival time)
Unavailability	Fraction of job downtime (1 - availability)
Dropped jobs	Fraction of jobs that have exceeded deadlines
ρ	Cluster load
C_A^2	Burstiness of job arrivals
Slowdown	Factor speed difference between running a job on preferred resources vs. non-preferred
Plan-ahead	Amount of time into the future that policies can plan schedules for.

Table 6.2: Metrics and parameters used in results section.

of a job given its server placement. In this simulation, we assume that servers as well as server racks independently fail for a configurable percentage of time. A job is considered temporarily unavailable any time k or more of its tasks are *simultaneously* unavailable. We picked $k = 3$ as it seemed reasonable for the number of servers in our high availability jobs.

Job **unavailability** (Table 6.2) is then defined as the average percentage of time that a given simulated HA job is unavailable (i.e., $1-$ availability). To approximate job unavailability, the simulator randomly selects failure times and computes the job unavailability. This process is repeated 10,000 times for each high availability job, and the resulting unavailability is then averaged across all high availability jobs within a trace. We have found this to produce stable results (error bars on Fig. 7.3(c) provide indirect support for that). Thus, the differences in availability between compared scheduling policies can be attributed to differences in their respective effectiveness.

To quantify the utility change associated with different levels of availability, a user is able to specify a discount factor as a function of unavailability. In our experiments, we use a basic function form of $\log_{10}(\frac{1}{unavailability})$, which has an asymptote at 0. This makes it increasingly more valuable to have lower unavailability (i.e., each additional "9" of availability has a big impact on the discount factor). We take this basic function form and scale it so that a placement yielding up to one machine per rack (roughly 10^{-5} unavailability in our setup) corresponds to no utility attenuation (i.e., scaling factor = 1.0). We also scale the curve so that having two machines per rack (roughly $3 * 10^{-4}$ unavailability, given our experimental configuration) has a discount

factor of 0.9.

6.2 Real Cluster

In addition to a series of simulation experiments, we conduct a series of real cluster experiments with TetriSched integrated in YARN 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. Hadoop YARN is chosen due to its increasing popularity as a mature open source resource management platform gaining both commercial support as well as adoption in large-scale production clusters, such as at Microsoft. We integrated TetriSched with the latest distribution of YARN at the time of the prototype development. We evaluate the performance of our Rayon/TetriSched stack relative to the mainline YARN Rayon/CapacityScheduler(CS) stack.

6.2.1 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 [24] 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.

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 6.3: Workload compositions used in results section.

6.2.2 Workload Composition

Workloads are often composed of a mixture of job types as the jobs vary in their preferences and sensitivity to deadlines. Table 6.3 shows the workload compositions used for experiments reported in this paper.

Load

Cluster load can be defined in one of the following three ways.

1. effective load—the actual work completed using cluster resources normalized by the total available cluster space-time capacity. This is often referred to as cluster utilization and is a factor $\rho_e \in [0; 1]$.
2. offered load—the total amount of ground truth work submitted to the cluster scheduler normalized by the available cluster space-time capacity. This is a factor that can exceed 1. Ground truth work is calculated as the amount of cluster space-time a job will consume if it's allocated preferred resources. We use this definition for simulation experiments and label it ρ (introduced in §6.1.3).
3. perceived load—the total amount of estimated work submitted to the cluster scheduler and normalized by the available cluster space-time capacity. This is work perceived by the scheduler and can significantly deviate from offered load due to runtime mis-estimation.

In our real cluster experiments, we report effective load ρ_e. In cases where real cluster experiments vary the extent of runtime mis-estimation, we report ρ_e statistic for the mis-estimate of zero. In simulation experiments, where the load is directly controlled as one of the experimental

control parameters, we report on the offered load ρ.

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 [15, 44, 59]. 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 [56, 59] 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. 3.4(a)).

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.

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

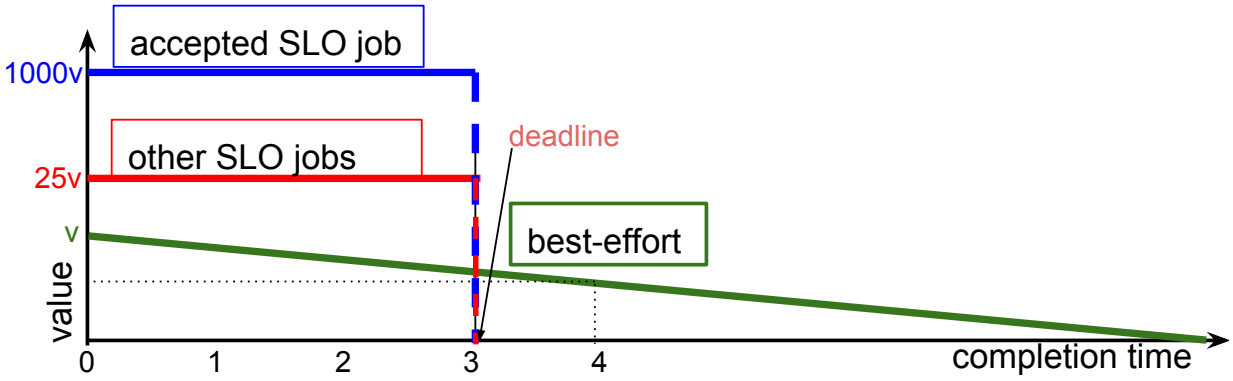


Figure 6.1: Internal value functions for SLO and BE jobs.

a reservation request to Rayon. In our experiments, a value function $v(t)$ encodes the sensitivity to completion time and deadlines (Fig. 6.1). 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.

6.2.3 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 in-

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 6.4: TetriSched configurations with individual features disabled.

creased 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 6.4)) to evaluate benefits from (a) soft constraint awareness, (b) global scheduling, and (c) plan-ahead by having each of these features individually disabled (Sec. 7.4.2). TetriSched-NG policy derives benefit from TetriSched's soft 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.

6.2.4 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 [8, 9], 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.

Chapter 7

Experimental Evaluation

In this chapter we validate both the practicality and the benefit of supporting declarative space-time soft constraints in cluster schedulers. The high level takeaway is that the flexibility of understanding and leveraging placement preferences both in terms of space (types and sets of resources) and time (when to start and finish) simultaneously allows a scheduler to exploit better information about each job's concerns and needs in both dimensions. As illustrated in Fig. 7.1, neither dimension alone is sufficient, with TetriSched outperforming the best space-only (alsched) and time-enhanced non-soft (Hard) options by 33% and 58%,[1] respectively. The three pairs of bars in Fig. 7.1 show the utility for schedulers that ignore constraints entirely (`None`), schedulers that accommodate hard constraints only (`Hard`), and schedulers that accommodate soft constraints ("Flexible").

This chapter is divided into two main parts: simulation and real system validation. We establish the benefit of space-time soft constraints first. In Sec. 7.1, we establish and empirically demonstrate that soft constraints improve the overall cluster scheduling quality, particularly as the heterogeneous cluster load increases. In Sec. 7.2, we add the temporal dimension, extending soft constraints to the two-dimensional cluster resource space-time. This is accomplished

[1] As we'll see in Section 7.2, TetriSched can outperform these other options by 3x or more under certain conditions.

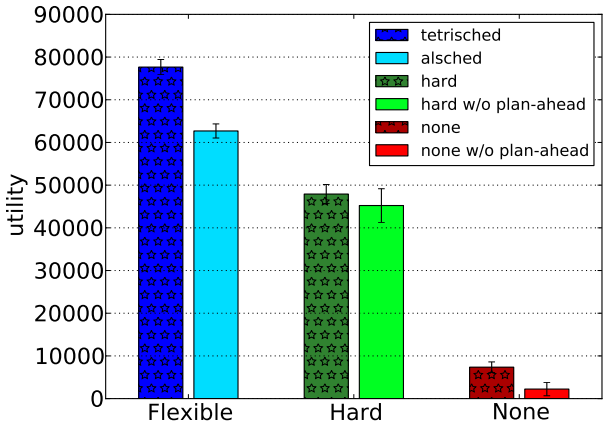
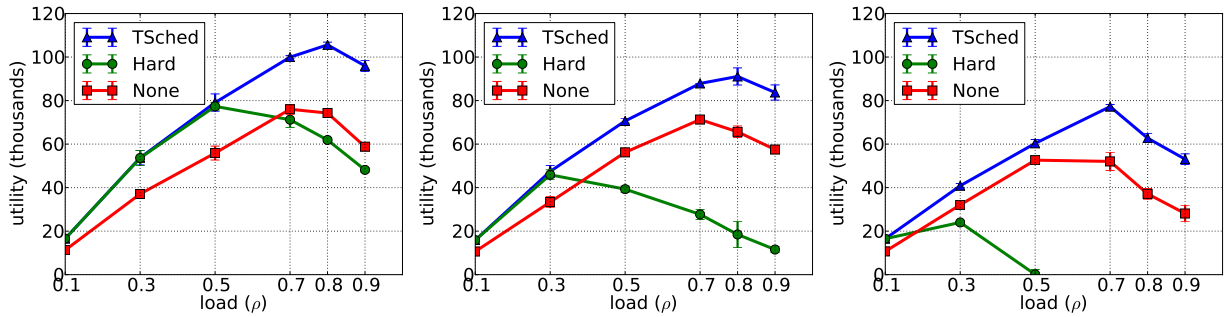


Figure 7.1: Better guidance leads to better scheduling. The three bar pairs correspond to schedulers that ignore constraints (`None`), that consider only hard constraints (`Hard`), and that consider soft constraints as well (`Flexible`). In each pair, the left bar exploits runtime estimates to plan ahead, while the right bar does not. The best option, by far, is `tetrisched`, which combines soft constraints with plan-ahead. Detailed explanation of how this data was measured and of the parameters used is provided in Sec. 6; the key parameters (for reference) are: workload mix=W2, plan-ahead=15min, slowdown=3, load $\rho = 0.8$, burstiness $C_A^2 = 8$, defined in Table 6.2.

with TetriSched's feature called plan-ahead (§5.2.1). Plan-ahead enables simultaneous consideration of all two-dimensional resource allocations, given job runtime estimates and possible job deadlines. Plan-ahead is shown to bring significant benefits in Sec. 7.2. We vary (a) the inter-arrival burstiness and (b) the strength of resource preference (slowdown) to show their effect. We transition to the second part of the evaluation chapter with a simulation study of TetriSched's sensitivity to runtime mis-estimation in Sec. 7.3.

Part two of this chapter focuses on the real system validation of our hypothesis. Sec. 7.4.1 focuses on the effect of runtime mis-estimation on scheduler performance in a real 256-node cluster. We demonstrate dramatic improvement in the number of jobs that satisfy their deadline SLO when scheduled with TetriSched as compared to YARN's default stack, particularly when job runtimes are under-estimated (Fig. 7.9(a)). We explore and separate the sources of this benefit in Sec. 7.4.2. The key Fig. 7.14(a) highlights the advantages of both space-time soft constraints and global scheduling for heterogeneous workload mixes in one graph. The takeaway result



(a) W2, slowdown=1.5 (b) W3, slowdown=1.5 (c) W4, slowdown=1.5

Figure 7.2: `TetriSched` outperforms `Hard` and `None` as cluster load(ρ) increases. Graphs 7.2(a), 7.2(b), and 7.2(c) correspond to workload compositions in Table 6.1 with a Poisson inter-arrival process ($C_A^2 = 1$) and schedulers using 15-minute plan-ahead.

is that each of our stated contributions improves the quality of scheduling decisions and must be used collectively for best performance. Sec. 7.4.3 concludes this chapter with a study of TetriSched's scalability in our real cluster.

Different workloads and systems used between different sections are carefully defined in §6 and explicitly stated for each set of results. Using a variety of cluster and workload compositions allows us to sweep a broader range of the parameter space, showing broader applicability of TetriSched's ideas. Note, though, that results for different workloads cannot be directly compared.

7.1 Spatial Preference Handling

In this section we illustrate the benefits of handling placement preferences as *soft constraints*.

Fig. 7.2 compares utility as a function of load (ρ) for the 3 workloads in Table 6.1. Workloads W2, W3, and W4 were chosen to create a progressively unbalanced load contribution coming from GPU, HDFS, and Availability jobs relative to the composition of the simulated cluster, configured to have 25% GPU nodes, 25% HDFS nodes, and 50% generic nodes. In all 3 of these figures, we see that `None` does worse than `TetriSched`. This is because `None` does

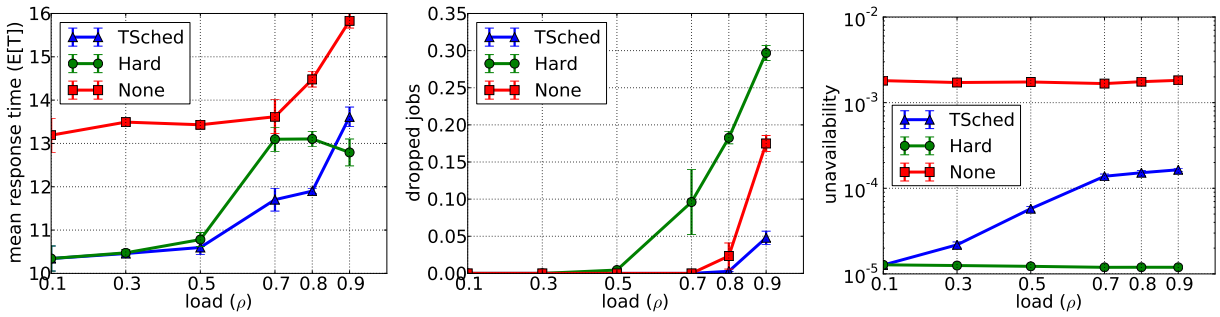
not consider what resources the workloads want; it treats all resources as equal and assigns them blindly, which causes the workload to run less efficiently.

When load is low, `Hard` performs similarly to `TetriSched` as there are enough resources to give most jobs their preferences. However, at higher loads when there is contention for resources, `Hard` is unable to consider alternative choices and is forced to wait. `TetriSched` performs much better since it is able to reason about these alternative choices when making scheduling decisions.

As we move from Fig. 7.2(a) to Fig. 7.2(b) to Fig. 7.2(c), the workload composition becomes more unbalanced, making it harder to find preferred resources. As a result, `Hard` starts dropping in utility at even lower loads. When the workload composition is unbalanced, `Hard` can actually do worse than `None` since it is restricted to only giving preferred resources, whereas `None` utilizes all resources albeit in a blind fashion. However, `TetriSched` is superior in all 3 scenarios since it can intelligently decide to use alternative resources when there is a resource imbalance.

7.1.1 Under the hood

What makes `TetriSched` yield higher utility in Fig. 7.2(a)? Fig. 7.3 shows the underlying metrics that affect the difference in utility. First, In Fig. 7.3(a), we see that the **average response time**, $E[T]$, (defined in Table 6.2) of `TetriSched` exceeds that of `Hard` for all but the highest load. Fig. 7.3(b) reveals that the utility difference is due to `Hard` dropping significantly more jobs. Recall that `Hard` will *always* cause jobs to wait for preferred resources, if not immediately available. When resources become available, `Hard` picks the youngest jobs to place on preferred resources as they have the highest value. As older jobs eventually reach their response time deadlines, they are dropped. `Hard` thus achieves good response times at the expense of dropped jobs when the load is high. `TetriSched` matches or exceeds the response time performance of `Hard`, but manages to drop a lot fewer jobs by exploiting the availability tradeoff for HA jobs (see Fig. 7.3(c)). Note that this tradeoff only occurs at high load when the cluster is most



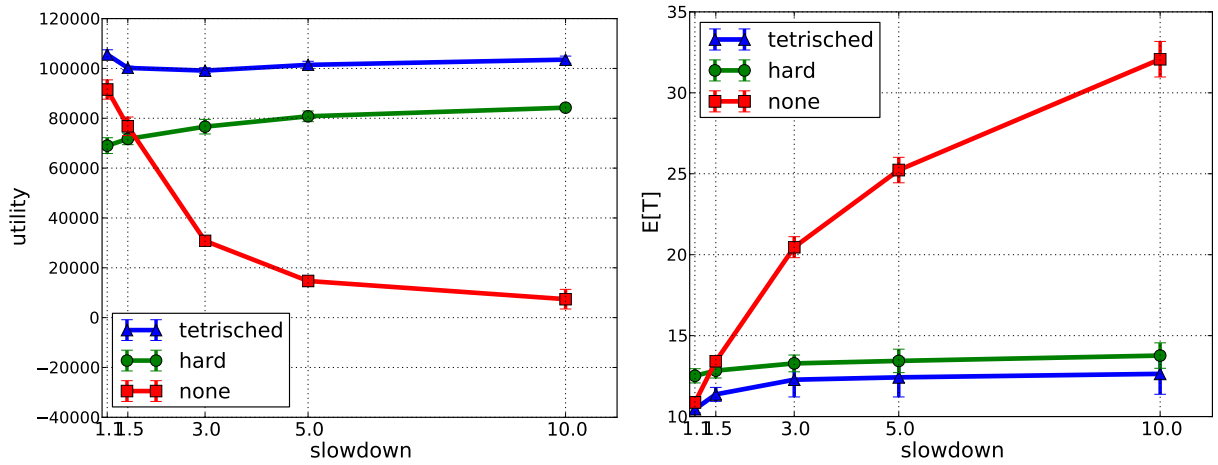
(a) mean response time (b) dropped jobs (c) unavailability

Figure 7.3: Under TetriSched more jobs meet the completion time SLO, while maintaining a response time comparable to Hard. Availability is reduced in preference to dropping jobs. None does worse on all metrics. Same setup as Fig. 7.2(a).

contended.

7.1.2 Effect of Slowdown

Slowdown (see Table 6.2) is an important factor that affects the relative performance comparison of the three spatial preference handling policies. For slowdowns as low as 1.1, None performs well (see Fig. 7.4(a)), since it doesn't suffer much from failing to prioritize preferred resources in the schedules it produces. As the slowdown increases, None starts performing increasingly worse, as the penalty for missing preferred resources increases with the slowdown factor on the x-axis. It thus passes Hard on its downward trend. Indeed, simply waiting for preferred resources becomes a reasonable scheduling policy when the benefits of doing so are orders of magnitude. TetriSched continues to outperform both of these policies, including and especially at all the intermediate slowdown values in this range (Fig. 7.4(a)). Lastly, slowdown affects TetriSched mean response time, $E[T]$, as well. Note that in Fig. 7.4(b) TetriSched actually surpasses Hard with respect to $E[T]$, particularly at slowdown factors of 3 or less.



(a) utility

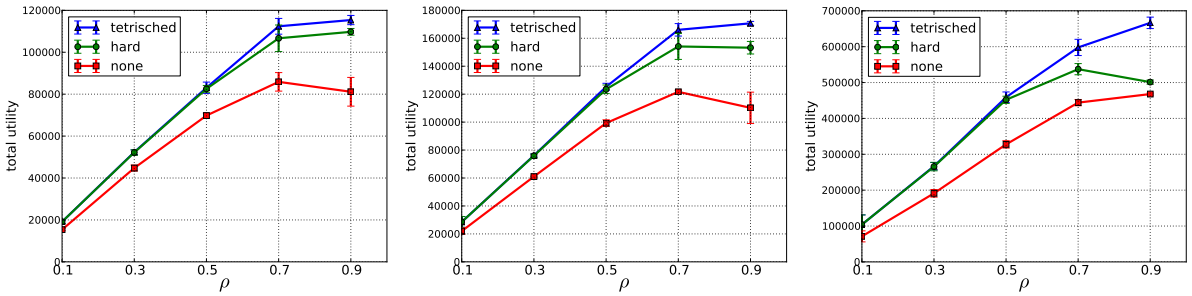
(b) mean response time

Figure 7.4: Utility and response time as function of slowdown. Workload is same as Fig. 7.2(a) at load ($\rho = 0.7$).

7.1.3 Handling Temporal Imbalance

In the perfectly choreographed match between resources and jobs, where each workload type can fit in its preferred cluster partition at full load, we expect the `Hard` policy to perform well (Fig. 7.5). It would, in fact, resemble the static partitioning allocation policy, with job sizes for each workload type carefully tuned to fit the corresponding partition on average. The key, however, is that job arrivals can be bursty and create transient *temporal* imbalances. Fig. 7.5 shows that spacial flexibility-aware scheduling policy handles such imbalances better than the alternatives. At low loads, temporal imbalances are absorbed by spare capacity in each of the preferred partitions, as each is overprovisioned. As load increases, however, `TetriSched` outperforms `Hard` and `None`, as transient overload is allowed to spill over into potentially available spare capacity in non-preferred partitions.

Lastly, we examine the effect of raising the priority of the "picky" jobs in this W1 mix. Recall that budget is the maximum utility a job can extract from running on the cluster. Relative budget differences, therefore, translate into relative job importance, as the scheduler favors jobs that yield higher utility. In Fig. 7.5(a), all jobs have the same budget. We increase the budget



(a) W1, 1x budget (b) W1, 2x budget (c) W1, 10x budget

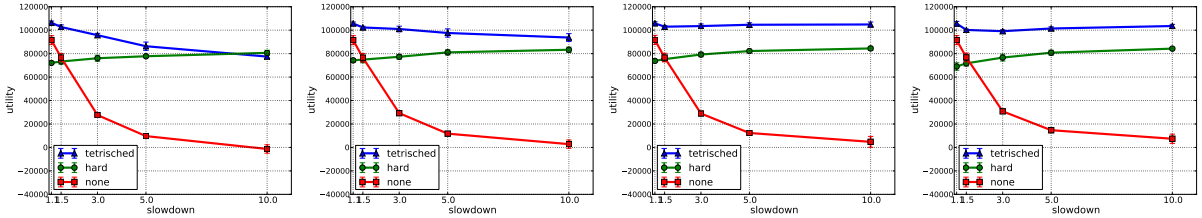
Figure 7.5: TetriSched leverages spacial flexibility, outperforming Hard and None through better handling of temporal imbalances. Increased importance of picky jobs leads to increased differential in performance, following Amdahl's Law.

for "picky" jobs by 2x in Fig. 7.5(b) and by 10x in 7.5(c) . As a result, TetriSched achieves higher relative gains when unconstrained jobs are less important than jobs with spacial preferences. Indeed, relative gains here are governed by Amdahl's Law—as the fraction of utility from "picky" jobs increases, so does the benefit of soft constraints.

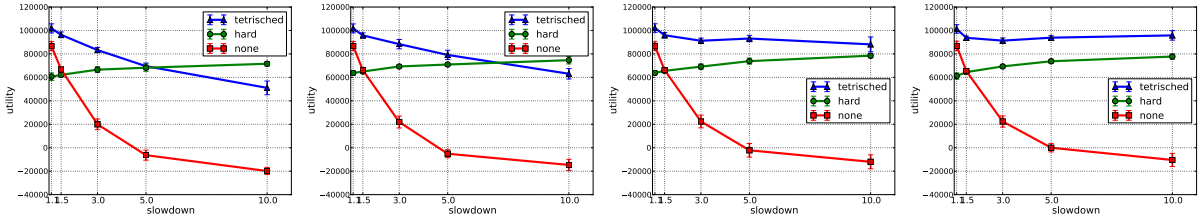
7.2 Benefit of Plan-ahead

A major feature of TetriSched is its ability to plan ahead in time, using future resource availability estimates as well as job delay sensitivity information. This section quantifies the benefit from the plan-ahead feature and shows that it is an important differentiator between TetriSched and alsched – the two policies that understand spacial flexibility.

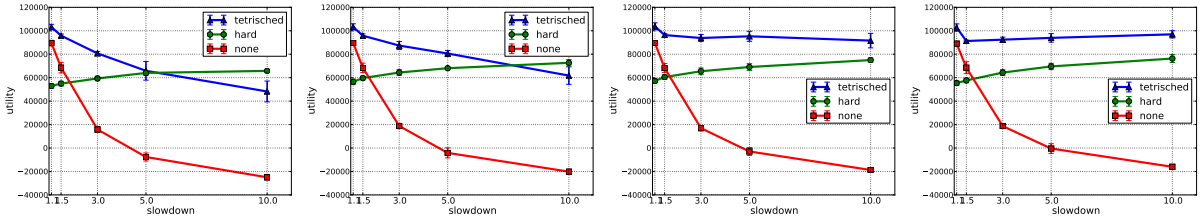
Fig. 7.6 plots utility for TetriSched, Hard, and None as slowdown increases from 1.1 to 10. TetriSched without plan-ahead (Fig. 7.6(a),7.6(e),7.6(i),7.6(m)), positioned in the first column of Fig. 7.6, represents the alsched system, which only understands soft constraints. We see that alsched starts making bad placement decisions relative to Hard at progressively higher slowdown factors because it does not understand the concept of waiting for preferred resources. As we go horizontally across this 4x4 grid, however, we see that TetriSched is



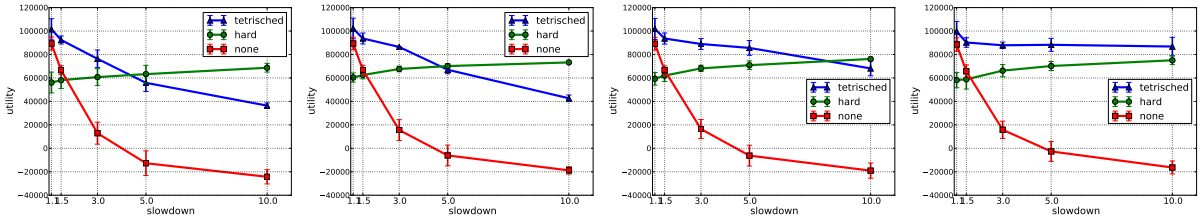
(a) no plan-ahead, $C_A^2 = 1$ (b) 2min plan-ahead, $C_A^2 = 1$ (c) 5min plan-ahead, $C_A^2 = 1$ (d) 15min plan-ahead, $C_A^2 = 1$



(e) no plan-ahead, $C_A^2 = 4$ (f) 2min plan-ahead, $C_A^2 = 4$ (g) 5min plan-ahead, $C_A^2 = 4$ (h) 15min plan-ahead, $C_A^2 = 4$

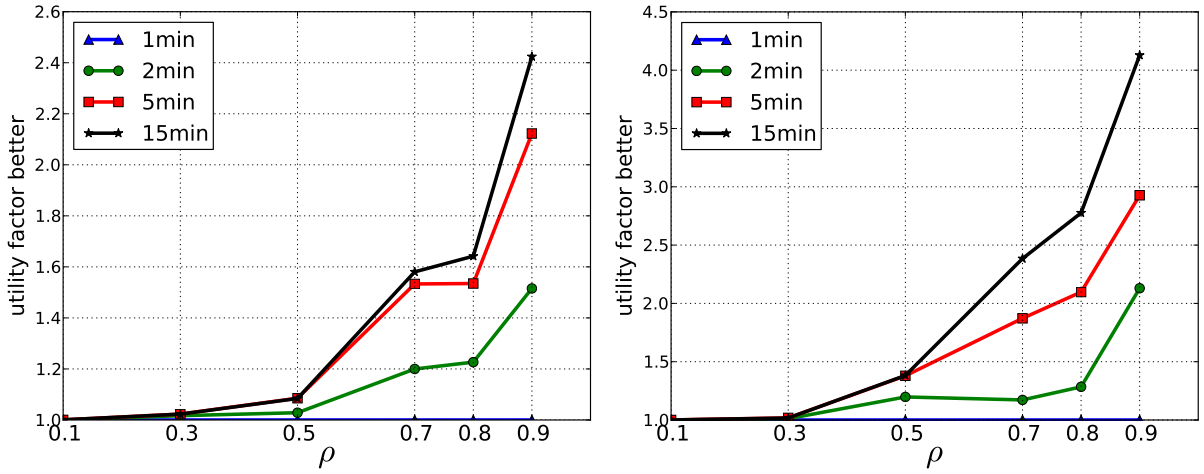


(i) no plan-ahead, $C_A^2 = 6$ (j) 2min plan-ahead, $C_A^2 = 6$ (k) 5min plan-ahead, $C_A^2 = 6$ (l) 15min plan-ahead, $C_A^2 = 6$



(m) no plan-ahead, $C_A^2 = 8$ (n) 2min plan-ahead, $C_A^2 = 8$ (o) 5min plan-ahead, $C_A^2 = 8$ (p) 15min plan-ahead, $C_A^2 = 8$

Figure 7.6: Time-aware scheduling is essential as slowdown increases. These graphs correspond to the W2 mix with a load ($\rho = 0.7$), horizontally varied plan-ahead, and vertically varied burstiness (C_A^2). We see that plan-ahead becomes even more important as the level of burstiness increases, particularly at high slowdowns. In fact, utility in this figure improves by a factor of upto 2.4x in going from no plan-ahead (`alsched`) to 15 min plan-ahead (`TetriSched`).



(a) slowdown=5

(b) slowdown=10

Figure 7.7: This graph shows factors of improvement for the TetriSched policy over alsched as a function of cluster load (ρ) and plan-ahead windows.

able to leverage plan-ahead to avoid this mistake and outperform both Hard and None, as the plan-ahead window increases from none to 15 min.

A natural question to ask is just how far ahead to plan. In Fig. 7.6(c) it may appear that a *smaller* plan-ahead window of 5 minutes might be sufficient. However, we also discovered that the positive effect of plan-ahead is significantly amplified by workload burstiness. As the temporal imbalance created by burstier workloads exerts more pressure on the cluster's scarce preferred resources, it becomes evermore important to leverage job runtime estimates and plan ahead pending job placement, instead of falling back to secondary options instantaneously. As we vertically trace Fig. 7.6(a),7.6(e),7.6(i),7.6(m), we observe a drop for alsched both in absolute utility and relative to Hard. The same downward trend can be observed, in fact, for any of the four subfigure columns of Fig. 7.6. Plan-ahead helps TetriSched handle this increasing temporal imbalance, however, illustrated with a gradually improving relative performance, as we horizontally scan any of the 4 subfigure rows. Fig. 7.7 highlights the factors of improvement TetriSched achieves from plan-ahead. Specifically, the highest factor difference for TetriSched between Fig. 7.6(p) and Fig. 7.6(m) is plotted in Fig. 7.7(b) as 2.4x ($\rho = 0.7$), and we see even higher

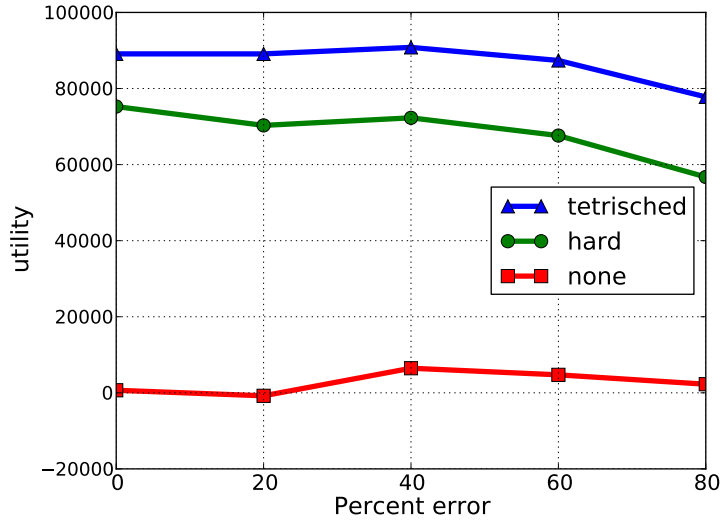


Figure 7.8: Effect on utility as users are more erroneous about duration estimates. `TetriSched` is robust to error in duration estimates. The average error is kept constant so that load ($\rho = 0.7$) remains constant. Percent error is calculated as the root mean square error divided by the average duration. This experiment uses the W2 mix with some burstiness ($C_A^2 = 4$).

factors of improvement for higher loads.

7.3 Sensitivity to duration mis-estimation

Users are unlikely to provide perfectly accurate guidance to the scheduler. This section evaluates the effect of inaccurate job duration estimates. Given that `TetriSched` uses plan-ahead, inaccurate job duration estimates could lead to bad scheduling decisions. But, somewhat to our surprise, we found that TetriSched's efficacy is robust to such inaccuracy. As expected, we found a qualitative correlation between situations for which plan-ahead matters the most and the utility drop-off as a function of inaccuracy (quantified as the coefficient of variance of root-mean-squared-error, or CV(RMSE)). In other words, duration estimate inaccuracies affect `TetriSched` only in cases where plan-ahead matters the most. But, the utility drop-off is small as we increase the coefficient of variance by as much as 0.8 of the mean job duration, which corresponds to 80% error on average. It's worth mentioning that perturbed runtime estimates could

deviate by as much as 3.4x. Figure 7.8 plots utility as a function of CV(RMSE). The dropoff in utility is insignificant until the CV(RMSE) of 0.6. `none` is least affected as it extracts the least benefit from plan-ahead, oblivious to benefits of preferred resources altogether. Utility difference for `TetriSched` was observed to be within 10-15% of perfect runtime estimates across a large range of workload parameters.

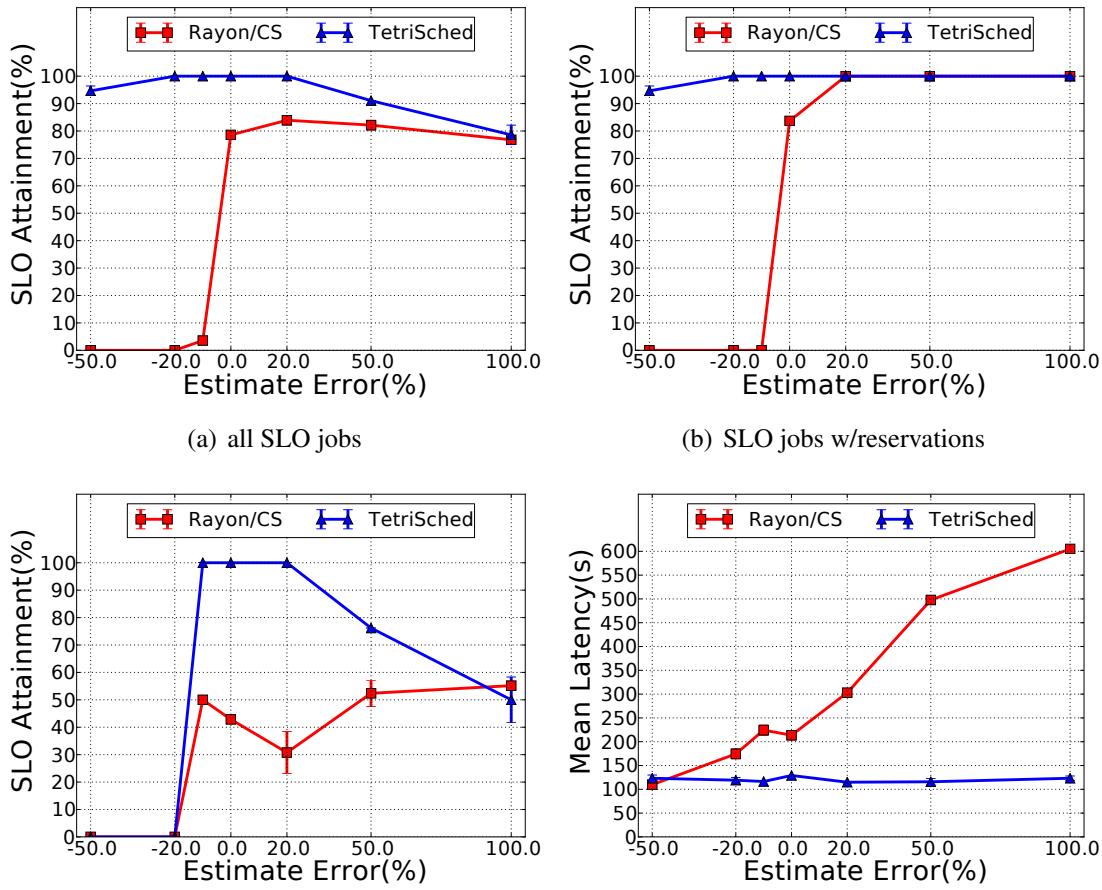
7.4 End-to-end evaluation on a real cluster

This section validates TetriSched, including its robustness to runtime estimate inaccuracy, the relative contributions of its primary features, and its scalability on a real cluster. 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 clusters in practice.

7.4.1 Sensitivity to runtime estimate error

Fig. 7.9 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



(a) all SLO jobs

(b) SLO jobs w/reservations

(c) SLO jobs w/o reservations

(d) Best Effort latency

Figure 7.9: 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. Rayon/TetriSched $\rho_e = 0.97$, Rayon/CS $\rho_e = 0.93$.

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

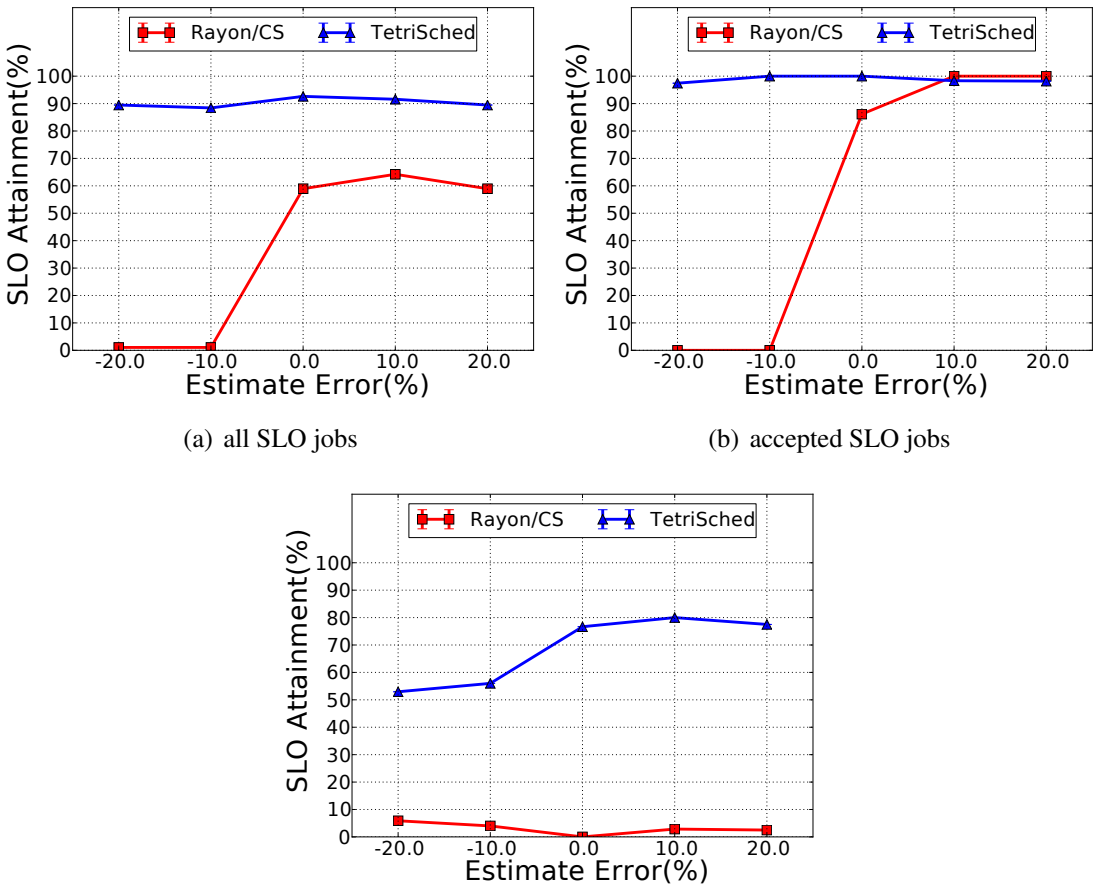
increases in best effort job latencies, because of increased pressure on the best-effort queue from the following main sources:

1. 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 used by YARN to guarantee capacity allocated to reservation queues for accepted SLO jobs.
3. the reservation system is in a higher state of flux due to over-estimation. As over-estimate-based reservations are released early, temporarily available capacity causes more best-effort jobs to be started. However, these jobs often don't complete before the next SLO job with a reservation arises, triggering preemption that wastes 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. 7.10 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 $\approx$100% SLO attainment for accepted SLO jobs.

7.4.2 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. 7.4.1 with a synthetic workload that is similar (homogeneous mix of SLO and best-effort jobs). As expected, we observe similar trends (Fig. 7.11), 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.



(a) all SLO jobs

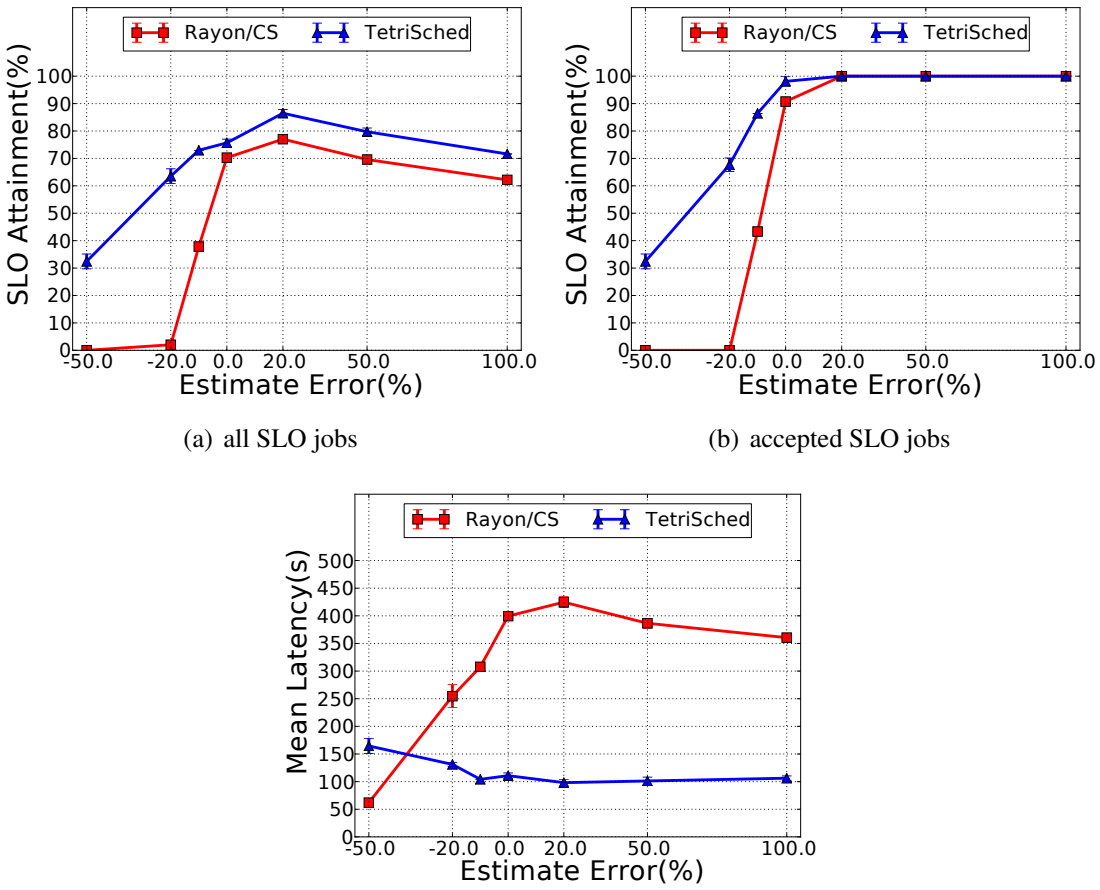
(b) accepted SLO jobs

(c) SLO w/o reservation

Figure 7.10: Rayon/TetriSched achieves higher SLO attainment for production-derived SLO-only workload due to robust mis-estimation handling. Cluster:RC256 Workload:GR_SLO. Rayon/TetriSched $\rho_e = 0.97$, Rayon/CS $\rho_e = 0.87$.

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. 7.12 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



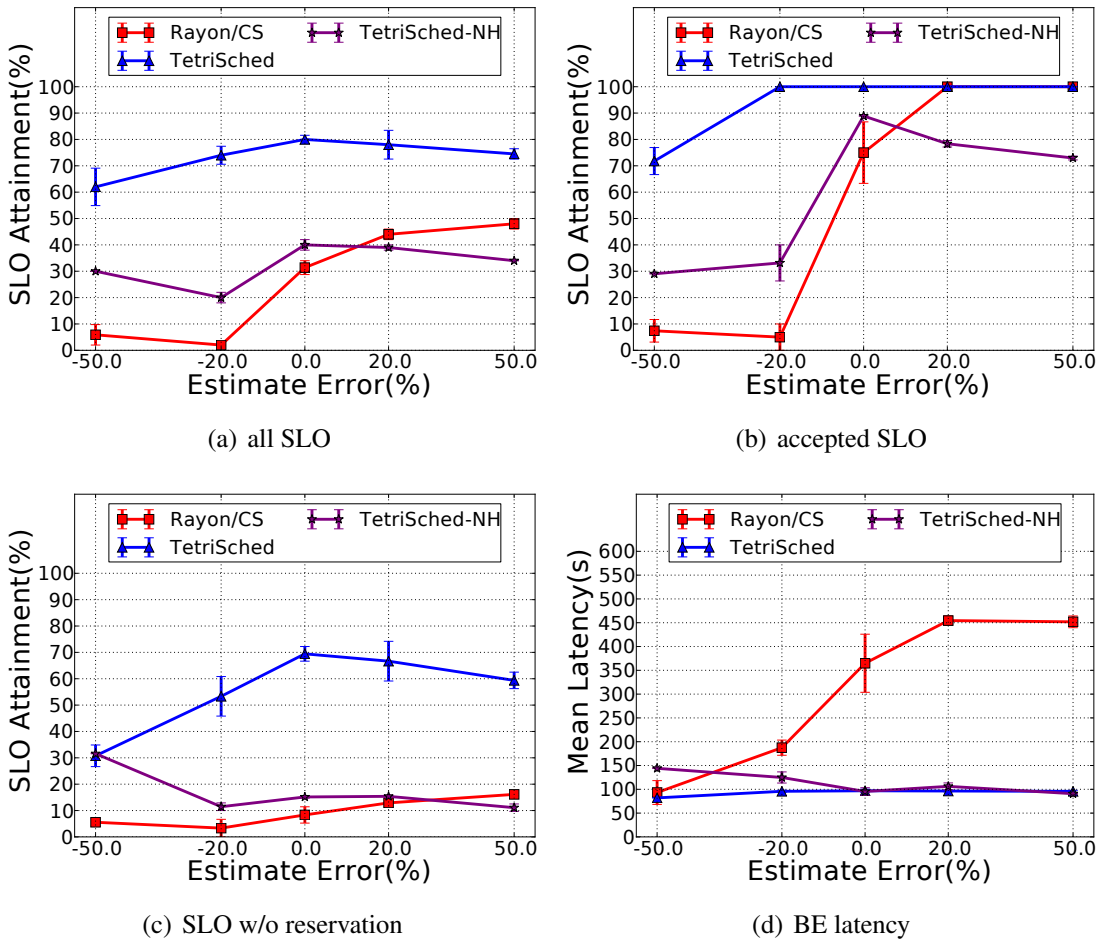
(a) all SLO jobs

(b) accepted SLO jobs

(c) latency

Figure 7.11: Synthetically generated, unconstrained SLO + BE workload mix achieves higher SLO attainment and lower latency with Rayon/TetriSched. Cluster:RC80 Workload:GS_MIX. Rayon/TetriSched $\rho_e = 0.93$, Rayon/CS $\rho_e = 0.919$.

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. 7.12(a)). Disabling soft constraint support can even be seen reducing the performance of TetriSched-NH below Rayon/CS as over-estimation increases (Figures 7.12(a) and 7.12(b)). While both Rayon/CS and TetriSched-NH are 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



(a) all SLO

(b) accepted SLO

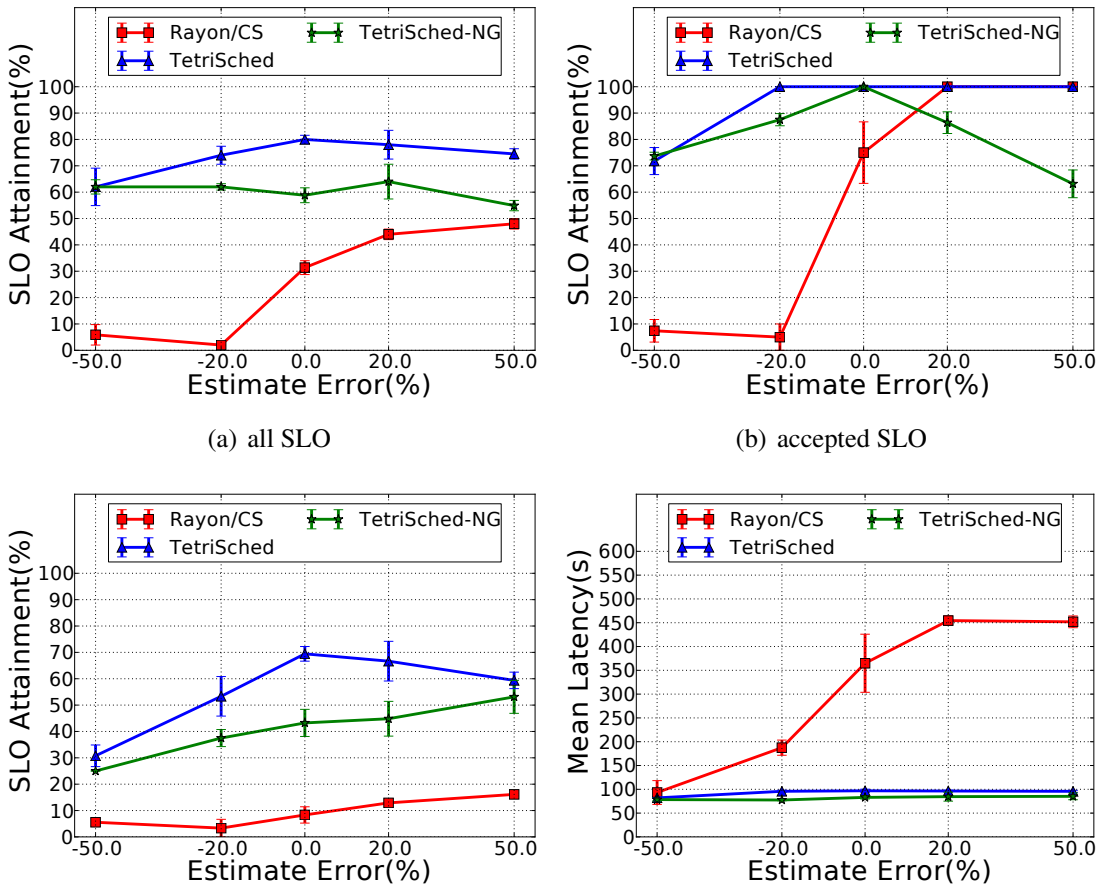
(c) SLO w/o reservation

(d) BE latency

Figure 7.12: TetriSched derives benefit from its soft constraint awareness—a gap between TetriSched and TetriSched-NH. Cluster: RC80, Workload: GS_HET. Rayon/TetriSched $\rho_e = 0.90$, Rayon/CS $\rho_e = 0.79$.

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. 7.4.3) of TetriSched's global scheduling, we introduce TetriSched-NG, our greedy scheduling policy (Sec. 6.2.3). It uses TetriSched full MILP formulation, but invokes the solver with just one job at a time, potentially reducing its time complexity. Fig. 7.13 compares TetriSched with a version using the greedy policy, referred to as TetriSched-NG, finding that global scheduling significantly increases



(a) all SLO

(b) accepted SLO

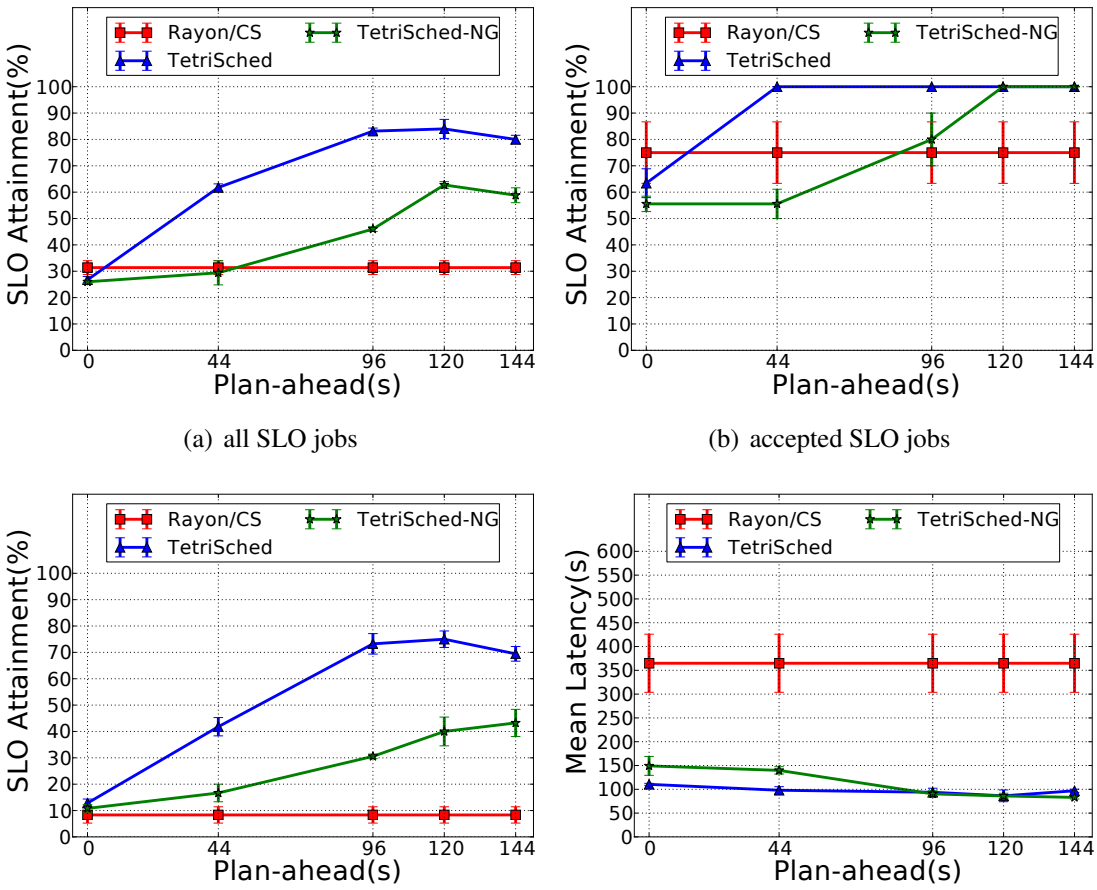
(c) SLO w/o reservation

(d) BE latency

Figure 7.13: 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. Load: Fig. 7.12.

SLO attainment. Global scheduling accounts for the gap of up to 36% (at 50% over-estimate) between TetriSched and TetriSched-NG (Fig. 7.13(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. 7.13(a)) and best-effort job latency (Fig. 7.13(d)), showing that greedy policies using TetriSched's other features are viable options if global scheduling latency rises too high.

Plan-ahead. Fig. 7.14 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-



(a) all SLO jobs

(b) accepted SLO jobs

(c) SLO w/o reservation

(d) latency

Figure 7.14: TetriSched benefits from adding plan-ahead to its soft constraint awareness and global scheduling. Cluster:RC80 Workload:GS_HET. Load: same as in Figures 7.12 and 7.13.

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. 6.2.3), which emulates the behavior of alsched [66]—our previous work. As we increase plan-ahead, however, SLO attainment increases significantly for TetriSched, until plan-ahead≈100s.

Summary. Fig. 7.12–7.14 collectively show that all three of TetriSched's primary features must be combined to achieve the SLO attainment and best-effort latencies it provides. Removing

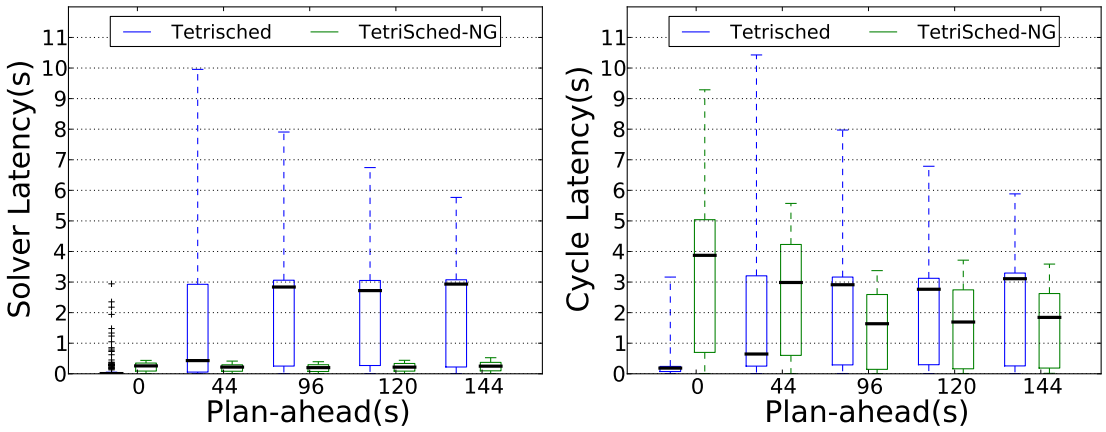
any one of soft constraint support, global scheduling, or plan-ahead significantly reduces its effectiveness.

7.4.3 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. 4) for TetriSched, as they are created per partition per cycle for each time slice of the plan-ahead window. Thus, in Fig. 7.15, we focus on the effect of the plan-ahead window size on the cycle (Fig. 7.15(b)) and solver (Fig. 7.15(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. 7.15(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.

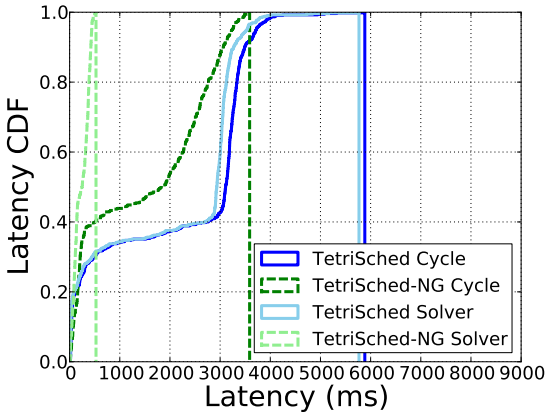
Fig. 7.15(b) reveals a surprising result: despite increasing the MILP problem size, increased plan-ahead can actually decrease cycle latency for the greedy policy. This occurs because scheduling decisions improve with higher plan-ahead (Sec. 7.4.2), reducing the number of pending jobs to schedule—another 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 vari-



(a) solver latency

(b) cycle latency



(c) CDF for plan-ahead=144

Figure 7.15: TetriSched scalability with plan-ahead.

ables [68]. Optimizations include extracting the best MILP solution after a timeout, seeding MILP with an initial feasible solution from the previous cycle (Sec. 5.2.6), 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 (§3.1.2)—all aimed at minimizing the resulting MILP problem size. Our simulation experiments in §7 show 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. 7.13),

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.

Chapter 8

Aramid: Impact in Production

To broaden the impact of our space-time soft constraint ideas, we collaborated with Microsoft on adapting a subset of them in YARN, as part of the next generation resource reservation framework called Aramid. The purpose of this work was to demonstrate that it is not necessary to fully replace an existing resource management infrastructure already in place in order to leverage some of the main ideas we propose with TetriSched. This technology transfer, in the process of being open sourced, also serves as the on-ramp to adopting a greater set of ideas and, ultimately implementing full support for our main contribution—*space-time soft constraints*—at the scale of a production cluster.

To accomplish this, we borrow two main TetriSched features: (1) heterogeneity-awareness and (2) soft constraints.

8.1 Heterogeneity in production

We describe three real scenarios and cumulative data from ten corporate datacenters to illustrate challenges.

CPU chipset impact on job runtime: This example was exposed by real user complaints regarding job runtime inconsistencies for a production Spark job. Figure 8.1 shows completion

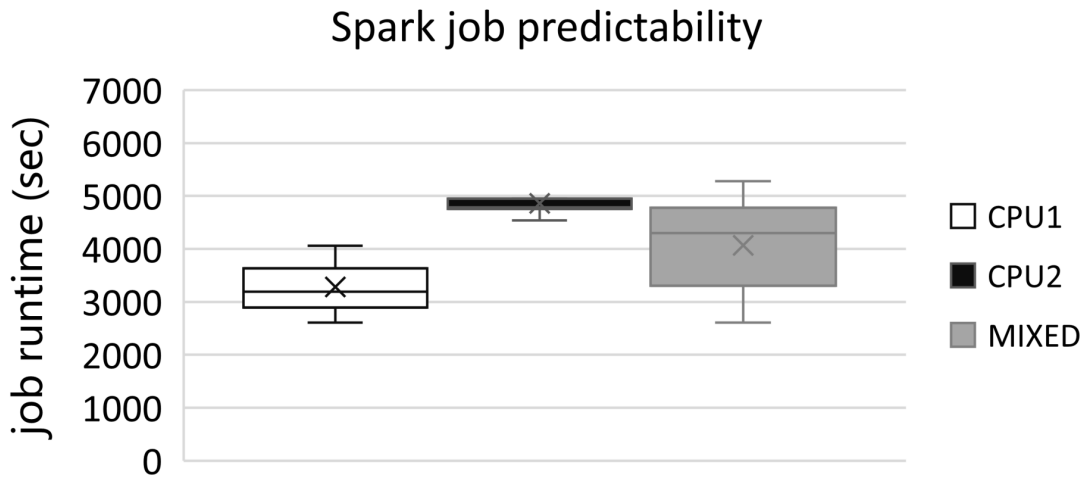


Figure 8.1: Runtime of a production Spark job on different CPUs (i.e., hardware architecture). The three boxplots represent 5 runs of this recurring job on CPU1 machines, 5 runs on CPU2 machines, and the 10 runs collectively.

times for a production batch Spark job on two generations of (nominally very comparable) CPUs that co-exist in one of Microsoft's large clusters. The left (white) and middle (black) rectangles represent the 25th to 75th percentiles of completion time on each of the two machine generations, for a number of runs of that job. The right (gray) rectangle shows the same percentiles when the job is scheduled on a randomly chosen set of machines, which would be the case in a system not explicitly aware of heterogeneity. The higher variation of the latter case induces production customers to conservatively over-reserve resources (lowering utilization) to manage risk of missed deadlines.

FPGA and Torus network for index serving: An important class of applications arises from hardware-acceleration technologies, where, e.g., a search-engine index can run on either standard CPUs or much more efficiently on FPGA-equipped nodes [54]. The FPGA machines are organized in a dedicated network. For example, in [54], every node in half a rack (48 machines) is directly connected by a network torus with peak bandwidth of 20Gbps. The FPGA-based solution requires "gang scheduling" of eight machines (i.e., allocations at multiples of eight) as eight FPGAs operating together as pipeline to serve each user query. In Fig. 8.2 we show that effi-

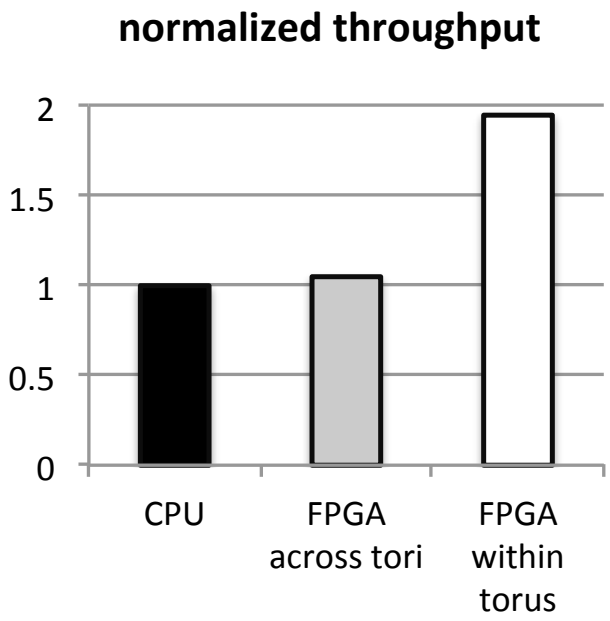


Figure 8.2: Normalized throughput achieved by index-serving services on different hardware configurations. This empirical data highlights the effect of interconnect topology on performance. Placement on FPGA alone yields similar performance to generic CPU resources. Only torus-local computation achieves $1.95\times$ throughput.

ciency gains can be substantial if *both* node *and* network heterogeneity are accounted for: $1.95\times$ greater throughput on the same number of machines[1]. If the allocation uses FPGAs without accounting for networking (e.g., an allocation of eight FPGAs, but four in each torus), the speedup obtained from specialized hardware is negligible. Another important aspect of this workload is its diurnal pattern with a peak-to-valley of over $3\times$. Any time-agnostic solution would be forced to over-provision significantly.

GPU and Infiniband for DeepLearning: The Yahoo! production DeepLearning pipeline [12] runs on heterogeneous Hadoop clusters with both standard CPU nodes connected via Ethernet top of rack switches and nodes equipped with 8xGPUs that communicate via RDMA over 100GBPs Infiniband. Data preprocessing for the pipeline can take place at any node, with a combination of Hadoop/Spark jobs, while the Caffe-based learning phase is much better served by the spe-

[1]This and other information regarding the Catapult project are derived from [54] and personal communications with its authors.

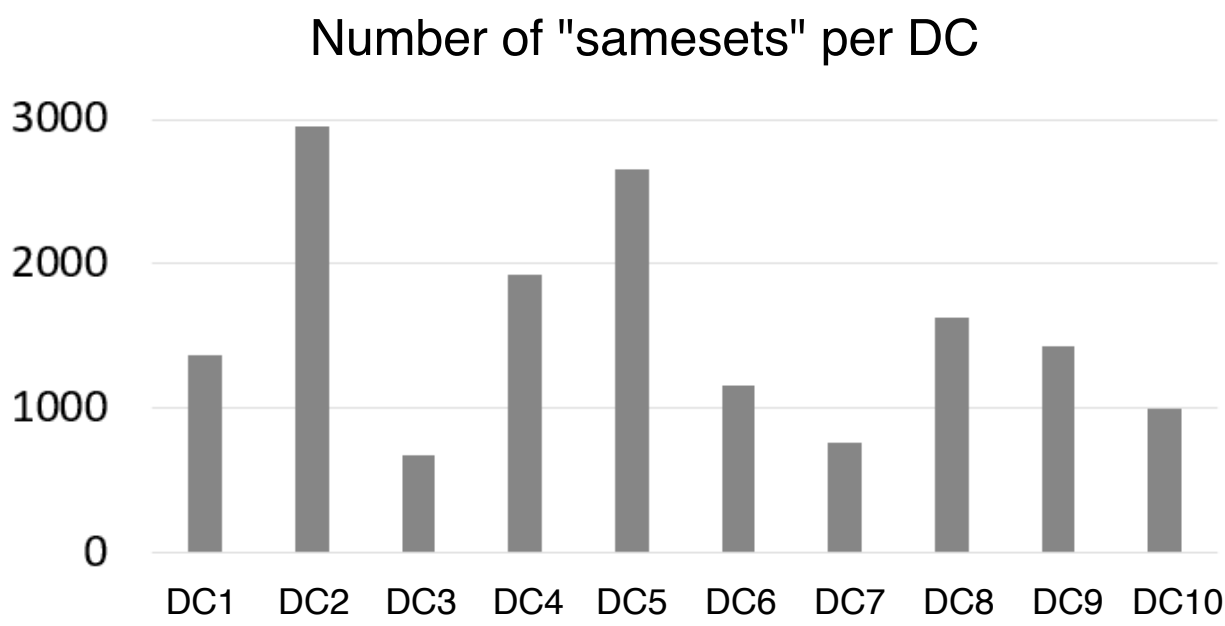


Figure 8.3: Number of samesets (i.e., groups of nodes with an identical set of accounted attributes, also—partitions) for each of ten Microsoft data centers.

cialized hardware—measurements [7, 27] show that Caffe runs 8–17× faster on GPUs than on regular CPUs. This scenario demonstrates the need to support pipelines that require different hardware in different phases of their execution.

Modern datacenters have substantial heterogeneity. We describe three specific examples of datacenter heterogeneity for concreteness, but modern datacenters exhibit a wide range of node heterogeneity. Figure 8.3 shows the number of samesets in each of ten Microsoft datacenters. (Recall that a "sameset" is a collection of homogeneous nodes within a heterogeneous cluster.) Each datacenter has 500–3000 distinct node types, as defined by physical and logical attributes of significance to jobs. While some of the samesets are small, some consist of 1000s of nodes, and all must be considered by a system servicing complex mixes of jobs with diverse constraints, preferences, and SLOs.

8.2 Aramid Architecture

We combine the time-varying capacity guarantees introduced by YARN's state-of-the-art reservation system [10] with TetriSched's declarative expression of cluster resource space. Specifically, we borrow TetriSched's heterogeneity-awareness and adapt the notion of space-time soft constraints it introduces. The end result—a system called Aramid—enables users to request cluster capacity guarantees on given types of resources, with declared *ordinal preferences*, and with variable demand specification over time.

Thus, we take the next stop in the historical evolution and enrichment of YARN's queue semantics. Initially (YARN[71] circa 2013 and earlier queue-based schedulers), YARN queues carried the semantics of infinite constant capacity guarantee, i.e. $c(t) = const$. Rayon [10] advanced the state of the art circa 2014 by making queues $c(t) \neq const$—a significant departure. With Aramid, we now take the next logical step, developing the notion of space-time capacity guarantees by vectorizing YARN queue capacity guarantee as follows:

$$c(t) = \vec{c}(t) = \begin{bmatrix} c_{p1}(t) \\ c_{p2}(t) \\ \cdots \\ c_{pn}(t) \end{bmatrix} \tag{8.1}$$

, where $c_{pi}(t)$ is queue capacity guaranteed at $time = t$ from partition pi.

As such, Aramid consists of the following principal architectural components (also shown in Fig. 8.4).

Language compiler translates time-varying resource demands specified with arbitrary boolean expressions on machine attributes. The compiler module converts these boolean expressions to a disjunction (an OR) of samesets. In Fig. 8.5, boolean expressions on machine attributes (e.g., CPU1) are shown in their equivalent disjunctive normal form (DNF) efficiently produced by the language compiler. It can be shown that any arbitrary boolean expression can always be reduced

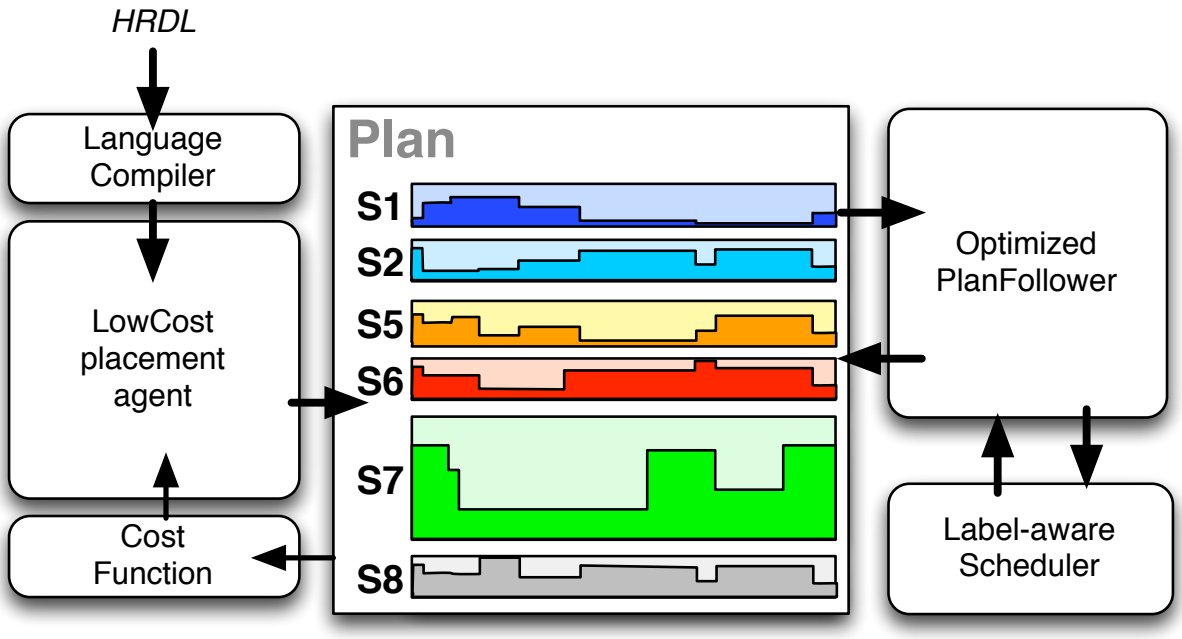


Figure 8.4: Conceptual view of Aramid's Architecture

to a DNF on samesets. This simplifies the design and implementation of the reservation plan and agent.

Reservation plan stores resource allocations across samesets and over time. The above compilation step allows us to track resources on a per-sameset basis. A key optimization enabled by the language compiler is to track resources only for "active" samesets (i.e., non-empty sets of one or more homogeneous machines). We find that, as cluster sizes scale, the number of active samesets remains small. This is a well contained number (<3000 samesets) even for our largest data center (Fig. 8.3). Reservation plan scalability is confirmed in §8.4.

`LowCost` placement agent allocates resource capacity across samesets within the specified time window of each reservation request. `LowCost`—the placement algorithm used—takes into account user allocation preferences. Placement decisions are driven by a configurable cost function that guides the search in the solution space. We chose a particular instantiation of that cost function which "balances" the load within and across samesets. This translates to smoother capacity allocations over time, which improves acceptance for future jobs with stricter requirements

and minimizes preemption of running containers. `LowCost` is described in detail in [11].

Optimized PlanFollower synchronizes the state of the reservation plan with the state of the underlying scheduler. It operates in a cyclic fashion. At each cycle, it takes a slice of all currently allocated reservations in the Plan and configures the underlying scheduler to honor those guarantees. Since this is done on a per-sameset basis, this module is a potential source of scalability challenges, which we successfully resolve by relaxing the unnecessary strong consistency guarantees inside YARN ResourceManager's queue maintenance logic.

Label-Aware Scheduler, configured by the PlanFollower, enforces allocated capacity guarantees on the right resource samesets. It manages the lifecycle of active jobs, cluster machines, their dynamically changing attributes and capacities. All sameset capacity and dynamic attribute changes are pulled periodically by the Planfollower and propagated to the Reservation Plan. Jobs are submitted directly to the scheduler with a reservation id acquired from the reservation agent and execute within the envelope of that reservation's guaranteed capacity. It is important to emphasize that this capacity potentially varies both over time and in the types of resources a job is guaranteed access to, in strict accordance with the user-specified HRDL expression.

8.3 HRDL

In this section, we first illustrate the Heterogeneous Reservation Definition Language (HRDL), by means of two examples, and then provide a formalization for it.

8.3.1 Example 1: Spark Job with allocation preference

We use the Spark job of Figure 8.1, which requires 80 containers each with 16GB and 8cores for 1 hour, if this reservation is mapped to machines of type "CPU1", or for 1.5 hours if it mapped to "CPU2" machines; a third option is to map the reservation to a mix of the two CPUs, where the minimum lease duration will be determined by the slower CPU (hence, 1.5 hours). While the

job can run on all three combinations above, it prefers to use the first option as its actual runtime is faster (hence, for example, less likely to stumble due to failures). The job also has some time flexibility, with a deadline of three hours from the submission time.

We use a script-like syntax and capture the above constraints in the following HRDL expression `rd` as follows:

```
b  = <16GB,8core>; // container size
h  = 80; // degree of parallelism;
g  = 80; // min parallelism (gang);
l1 = 1hr; // min lease duration;
w1 = 80container/hours // total work to place
q1 = "CPU1"; // node-label expression
a1 = ATOM(b,g,h,l1,w1,q1); //atomic expression

l2 = 1.5hr;
w2 = 120container/hours // total work to place
q2 = "CPU2";
a2 = ATOM(b,g,h,l2,w2,q2);

q3 = "CPU1 or CPU2";
a3 = ATOM(b,g,h,l2,w2,q3);

p  = P_ANY(a1,a2,a3); // preferential ANY
s  = now();    // start time
d  = s + 3hr; // deadline
rd = WINDOW(p,s,d); // time window
```

The example highlights how the user can specify the time window for resource allocation (via the `WINDOW` predicate), how the system could choose amongst a rank-ordered list of alloca-

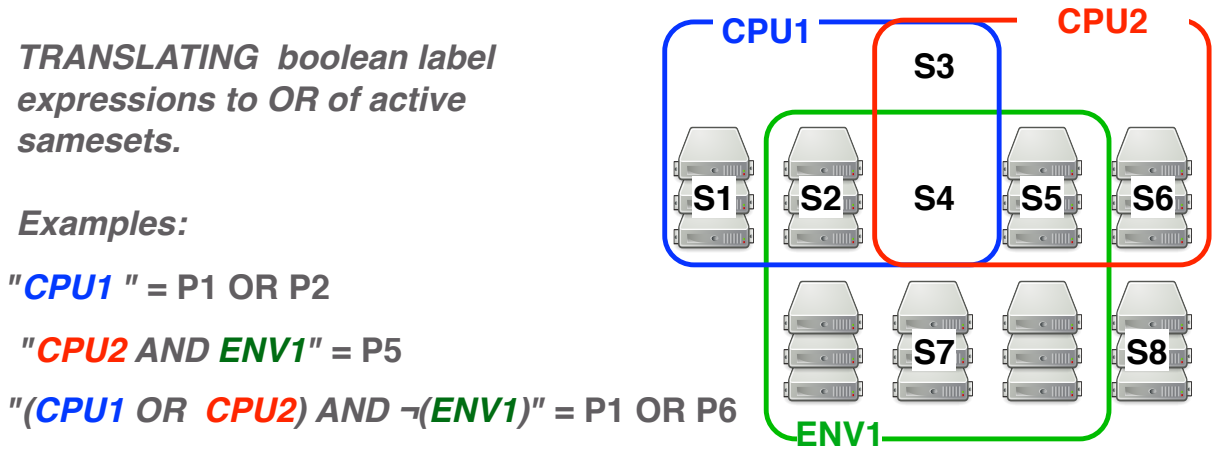


Figure 8.5: HRDL Compiler translates arbitrary attribute-based boolean expressions into a union of a subset of partitions.

tion alternatives (via the `P_ANY` predicate), and precise choices of parameters for an individual allocation (via the `ATOM` predicate).

8.3.2 HRDL Formalization

We develop HRDL by extending Rayon's resource definition language described in [10]. Our extensions enable users to explicitly specify heterogeneity in resource needs as well as ordinal allocation preferences. For the sake of completeness, we provide a formalization of the language from [10], adapted to include the novel extensions.

An HRDL *expression* can be:

An **atomic expression** of the form `ATOM(b,g,h,l,w,q)`, where: b is a multi-dimensional bundle of resources[2] (e.g., <2GB RAM, 1 core>) representing the "unit" of allocation (i.e., container), h is the maximum number of containers the job can leverage in parallel, g is the minimum number of parallel containers required by the job; a valid allocation of capacity at a time quanta is either 0 containers or a number of containers in the range $[g, h]$. l is the minimum lease duration of each allocation; each allocation must persist for at least l time steps, and w is

[2]This matches YARN containers [71] and multi-resource vectors in [21].

the threshold of work necessary to complete the reservation (expressed as container hours); the expression is satisfied iff the sum of all its allocations is equal to w. *The last parameter q encodes heterogeneity*. It is an arbitrary boolean expression of labels (e.g., "`(CPU1 or CPU2) and !ENV1`"), and bounds this atom to be satisfied over a certain set of machines, i.e., the ones which carry a set of labels matching this expression.

A **choice expression** of the form `ANY(`$e_1, \ldots, e_n$`)`. It is satisfied if *any* one of the expressions e_i is satisfied.

A **preferential choice expression** of the form:
`P_ANY(`$e_1, \ldots, e_n$`)`. It is satisfied if *any* one of the expressions e_i is satisfied. In particular, considering resource heterogeneity, order matters, and the planning system is asked to satisfy the e_i with the lowest index i possible.

A **union expression** of the form `ALL(`$e_1, \ldots e_n$`)`. It is satisfied if *all* the expressions e_i are satisfied.

A **dependency expression** of the form `ORDER(`$e_1, \ldots, e_n$`)`. It is satisfied if for all i the expression e_i is satisfied with allocations that strictly precede all allocations of e_{i+1}.

A **window expression** of the form `WINDOW(e,s,f)`, where e is an expression and $[s, f)$ is a time duration interval. This bounds the time range for valid allocations of e.

We note that in the items above, each e_i can be an atomic expression, or, more generally, could nest other operators. For example, extending example 2, one may have ORDER(c, P_ANY(m1,m2)), where $m1$ and $m2$ represent two different alternatives for running the ML-stage, say on FPGA or or regular machines – for longer duration).

It is easy to see that HRDL allows users to express completely *malleable* jobs such as MapReduce (by setting $g = 1$ and $l = 1$) and very *rigid* jobs such as MPI computations requiring uninterrupted and concurrent execution of all their tasks (by setting $g = h$ and $l = w/h$). The `WINDOW` operator allows to constrain the interval of validity for any sub-expression. Its natural application is to express completion deadlines. Users can represent complex pipelines and DAGs

of jobs in RDL using the `ORDER, ALL`. The `ANY` operator allows one to express alternative options to satisfy a single reservation.

The key extensions we provide are the ability to constrain each `ATOM` to a specific set of machines, and the ability to impose a ranking order among alternative options via the new operator `P_ANY`. These syntactic extensions, albeit small, substantially increase the expressivity of the language.

Given a set of HRDL expressions which arrive online, the role of the planning phase is to examine if and how to accommodate each request. We next describe the underlying algorithms for making these decisions.

8.4 Aramid: Experimental Evaluation

Finally, we validate Aramid with production-derived workloads both on two physical clusters (a 265-node and a 2700-node production cluster) and in simulation. Recall that the two major goals were (a) to validate TetriSched's space-time soft constraint ideas in production and (b) to perform validation at production scale and with production or production-derived workloads.

First, we show that the combination of heterogeneity-awareness and allocation preference support yields 43% more goodput on production-derived cluster workloads on a real cluster (Fig. 8.6). Second, we conduct thorough scalability tests in simulation as well as on a 2700-node production cluster (§8.4.4).

8.4.1 Experimental Setup

We now describe the systems we compare, metrics of success, workloads, and cluster configurations.

As our baseline, we use a recent version of Apache Hadoop YARN [71] (trunk as of July 2015). Aramid's best configuration (Aramid-LowCost) is then compared against alternative

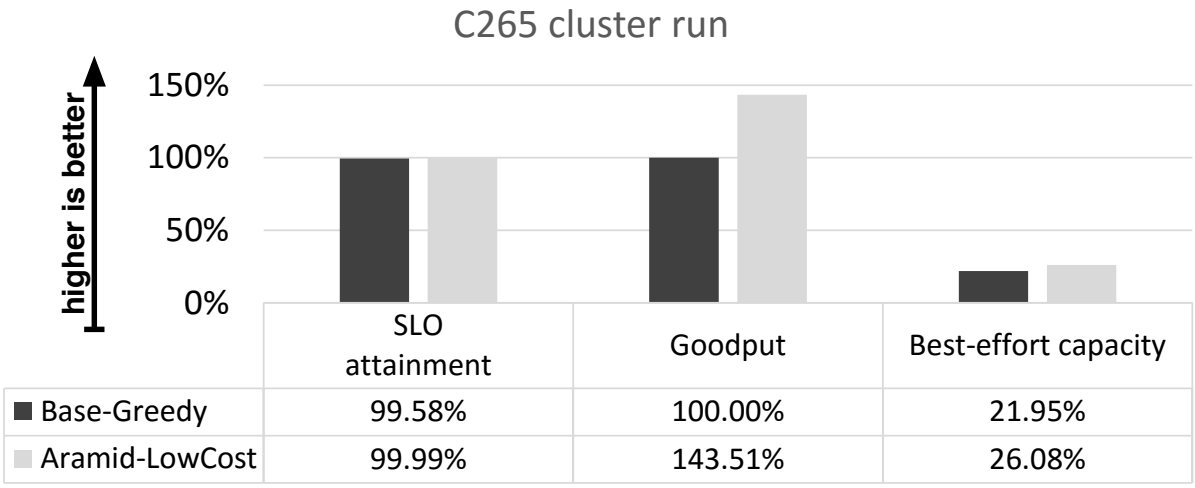


Figure 8.6: Aramid-LowCost achieves higher goodput, near perfect SLO attainment, and uses less resources on a 265-node cluster with Prod1 workload.

Base-Greedy	YARN [71], with Greedy [10]
Base-LowCost	YARN [71], with LowCost algorithm
Aramid-Greedy	Aramid, with label-aware Greedy
Aramid-LowCost*	Aramid, with label-aware LowCost

Table 8.1: Systems compared for Aramid evaluation.

Aramid and baseline configurations in Table 8.1.

Throughout our experiments we compare the above systems according to the five metrics described in Table 8.2. These metrics combined provide insight into the system's predictability, and efficiency (work delivered / used resources).

Unless otherwise specified, our experiments are based on a workload trace derived from a

Success Metrics

SLO attainment	percentage of jobs receiving all of the resource they reserved
Goodput	sum of good work performed (cpu-hours)
Acceptance	count of accepted reservations
Best-effort Capacity	resources left to best-effort jobs
Preemption	potentially preempted containers

Table 8.2: Aramid success metrics

Prod1 Workload

framework	class	freq. %	avg duration	avg parall.	alloc. pref.
MR/TEZ	S	7%	73	1.5	none
	M	15%	156	19	
	L	0.6%	2778	469	
SPARK	S	39.8%	173	2.6	soft
	M	14.52%	605	18	
	L	7.8%	1400	88	
	XL	4%	6300	510	
	XXL	8.6%	24570	1000	
MPI	-	1.56%	7800	400	hard

Table 8.3: Prod1: composition of a production workload extracted from a 4k cluster.

4k-node cluster at Microsoft. We refer to it as **Prod1** and describe its key characteristics in Table 8.3.

We categorize the jobs in this workload in 9 classes based on framework (MapReduce/Tez, Spark, MPI) and job size. We then extract several key statistical distributions: job arrival times, workload frequency, job parallelism, job duration, and allocation preference type.

Reservation deadline data was not readily available. Nevertheless, based on conversations with internal users and cluster operators, we choose the deadline to be normally distributed around 1.5 times the duration of the job. We determine the relative execution runtimes across heterogeneous labels based on job profiling (e.g., Figure 8.1). We then use standard Hadoop tools (Gridmix [64]) to generate thousands of reservation and job submissions, driven by the above distribution parameters.

8.4.2 Comparing with the State of the Art

In this section, we compare our system (Aramid-LowCost) with a state-of-the-art baseline (Base-Greedy) running on a real cluster.

265-nodes cluster run. To measure the effect of heterogeneity- and preference-awareness HRDL provides in Aramid, we ran a long experiment on a 265-node cluster with the Prod1 workload,

and report on 4h of steady-state execution. Figure 8.6 shows that Aramid-LowCost achieves 43% higher goodput, while leaving slightly more capacity for best effort jobs and matching baseline's SLO attainment. The Base-Greedy baseline is not heterogeneity-aware. Thus, it ignores MPI class placement constraints, scheduling those jobs anywhere. Further, Base-Greedy does not support allocation preferences. Therefore, it is *forced to conservatively estimate job runtimes* in order to retain good SLO attainment. Aramid, however, represented by Aramid-LowCost in Fig. 8.6, understands both heterogeneity and allocation preferences. It, thus, accommodates constraints specified by both the MPI and SPARK workload classes and is able to draw much tighter reservation envelope around jobs, as it controls explicitly on which type of nodes they will run. The SLO attainment results in Fig. 8.6 are equivalent (Aramid-LowCost achieves a marginal advantage). However, resource efficiency results are overwhelmingly in favor of Aramid-LowCost. Aramid delivers 43.5% higher goodput with about 4% less resources. This increased efficiency, translates in higher ROI for large production clusters.

8.4.3 Benefit from heterogeneity awareness and ordinal preferences

First, a substantial amount of performance benefit comes from support for heterogeneity- and preference-awareness in Aramid. We confirm this with the following two simulation experiments. First, we repeat the real cluster experiment in Fig. 8.6 and obtain consistent results for Base-Greedy and Aramid-LowCost. In this run, we also simulate Base-LowCost and Aramid-Greedy, finding that the Prod1 workload (as opposed to [8, 9]) has little flexibility—most jobs are rigid gangs. This bounds possible algorithmic gains. Most of improvement comes from HRDL expressivity.

Second, we explore the potential of HRDL when faced with even more heterogeneity, by simulating the FPGA index-serving workload sharing a heterogeneous cluster with our Prod1 workload. We simulate a 2700 node cluster with 1632 FPGA host nodes connected via tori (as in [54]) and 1068 regular nodes. Such a cluster composition is sufficient to accommodate a

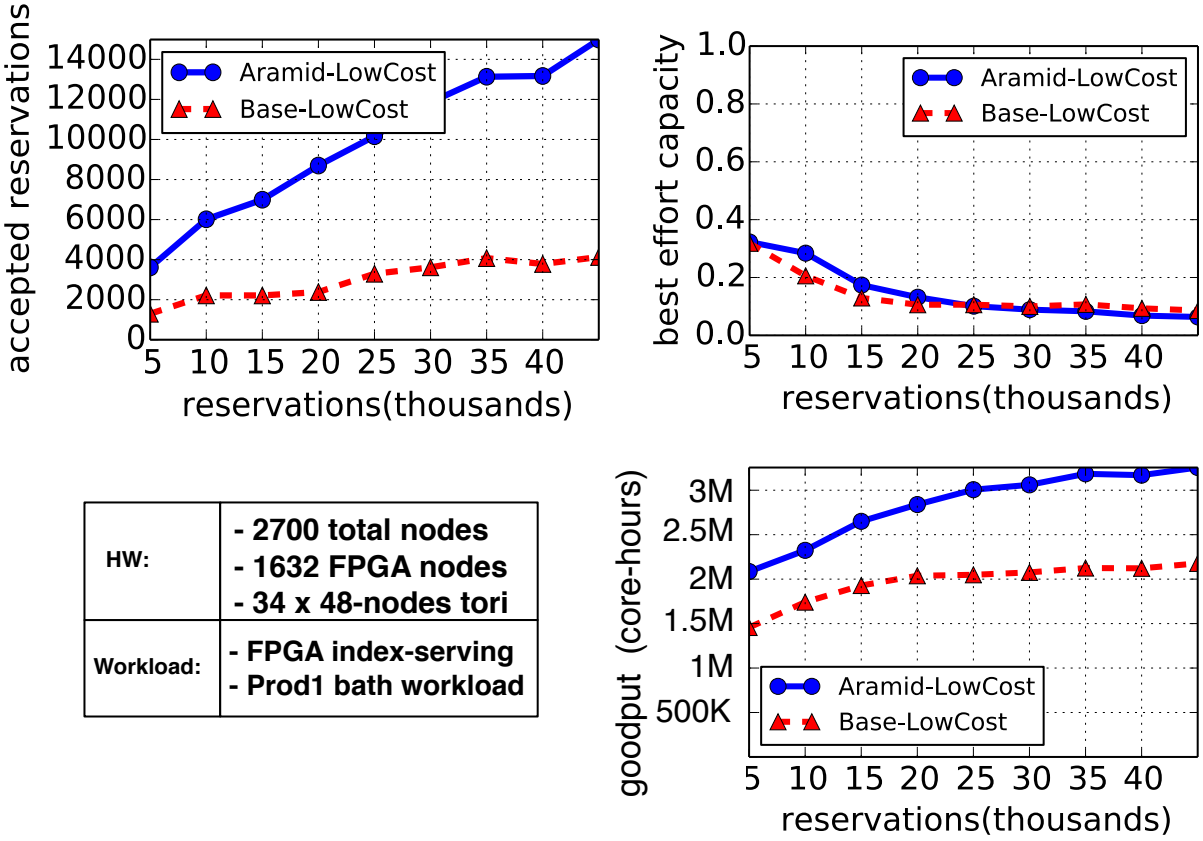


Figure 8.7: Aramid achieves higher goodput, accepts more reservations into the plan, and leaves a comparable amount of best effort capacity.

heterogeneity-unaware Base-LowCost (which is less efficient and needs more nodes). We automatically[3] derive HRDL expressions from the production traces used for [54] for both Aramid-LowCost and Base-LowCost and scale them to fit this simulated cluster.

Results (Fig. 8.7) indicate that HRDL-enabled Aramid-LowCost accepts up to $3.5\times$ more reservations than the LowCost-enabled baseline and increases cluster goodput by up to 50%, while leaving a comparable amount of capacity for best effort jobs. The ability of HRDL to model preferences, allows us to pick the preferred hardware for the serving workload, as well as for the Spark jobs in Prod1 whenever possible, but also spill over to less desired hardware when needed, thereby packing the cluster more tightly.

[3]By means of tooling available internally at Microsoft.

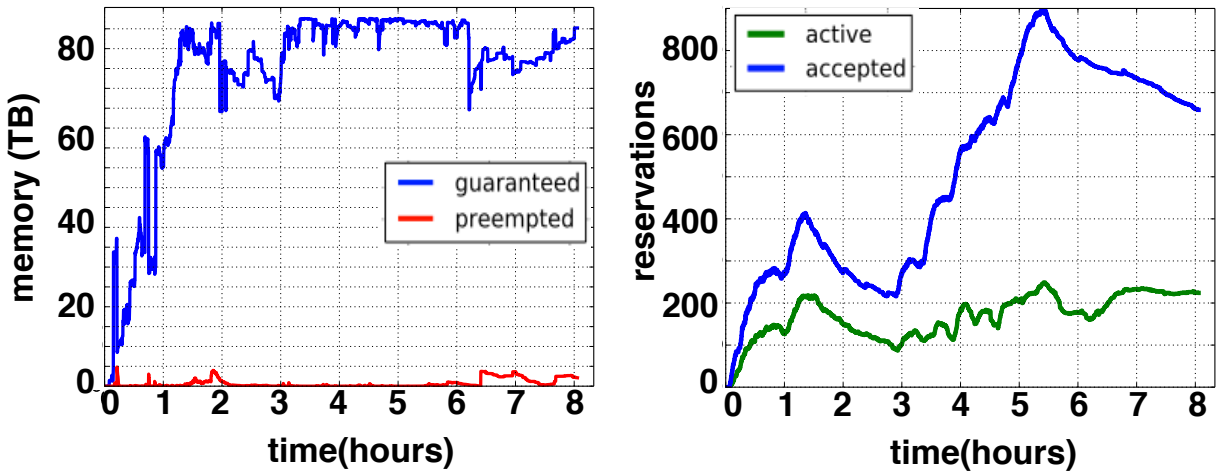


Figure 8.8: Scalability metrics for large scale *real cluster* run (on 2700 nodes)

8.4.4 Aramid scalability and practicality

In this subsection, we confirm Aramid's scalability both on a real cluster and simulation.

Aramid scales to 2700 nodes. Lastly, we validate Aramid's scalability to target production clusters, by running it live on a *large production cluster* with tens of labels on 2700 machines, scheduling almost 100k concurrent containers through the ResourceManager. We run a sustained 8 hours experiment, with hundreds of reservation submissions per hour. We measure the system performance both as perceived by the user (not shown), and as observed by instrumented system components (Fig. 8.8).

The key takeaway of this experiment is three-fold. First, we demonstrate that Aramid is able to sustain high load on a large cluster. Second, we show that Aramid can achieve high plan utilization. Third, we confirm that user-facing latencies are in-line with production cluster user expectations. We see up to 900 concurrent reservations in the plan, with up to 270 of them active throughout the 8hr run. At peak, aggregate guaranteed capacity exceeds the 92TB of container memory, reaching maximum plan capacity. The system remains responsive throughout the experiment with reservation submission latencies within 10sec. Internally, we measure the key PlanFollower latency. It is one of the most loaded components of the system, as it per-

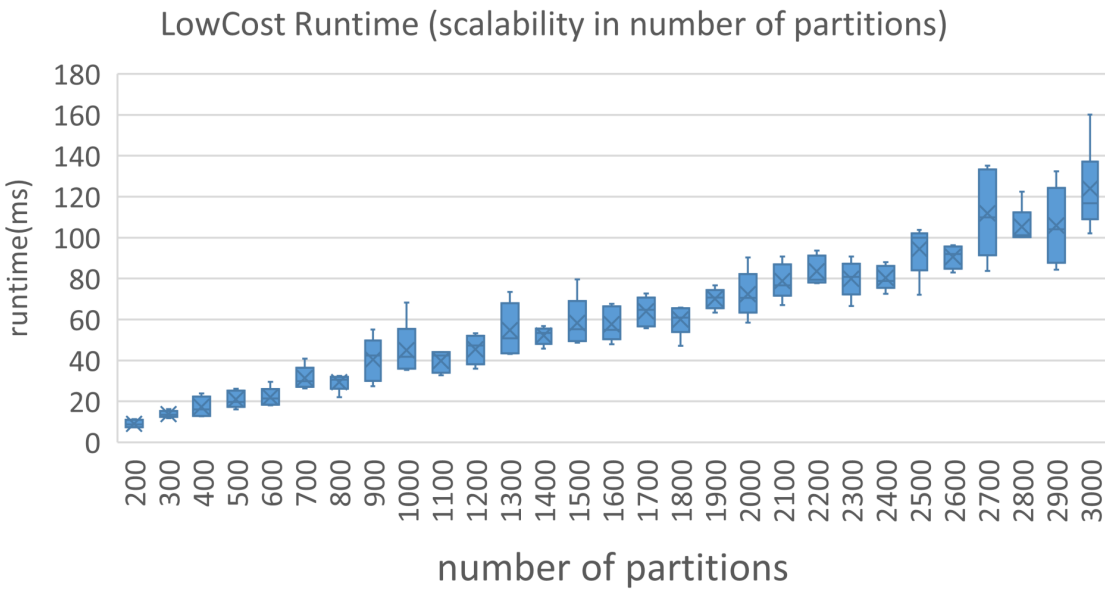


Figure 8.9: Aramid scales *linearly* up to 3000 samesets. Submitted HRDL expressions randomly choose and permute $X/2$ samesets, increasing load on the LowCost placement agent and space requirements in the Plan.

forms $|reservation| * |samesets|$ operations per its cycle, while synchronizing the plan with core scheduler data structures (§8.2). Despite the load, the Planfollower latency is well below its configured 1sec cycle period. Our implementation derives benefit from improved scheduler locking mechanisms and outperforms our initial implementation by an order of magnitude in the number of concurrent reservations and two orders of magnitude in worst-case cycle latency.

Aramid scales to 3000 samesets. In simulation, we stress test Aramid's LowCost placement agent and the reservation plan (Fig. 8.4)—its core algorithmic components. In Fig. 8.9, we submit 10k reservations for each data point and report average reservation submission times. We sweep 100 to 3000 sameset range to match our data center (Fig. 8.3). Submitted HRDL expressions randomly choose and permute $X/2$ samesets to progressively increase the load on the LowCost placement agent and the space complexity of the Plan. We definitively conclude that Aramid scales *linearly* up to our target datacenter sameset counts.

8.5 Other Impact

Our work was featured in a recent presentation [2] by Raghu Ramakrishnan, CTO for Data and Technical Fellow at Microsoft. Anecdotally, some of the biggest and most immediate impact was perceptual—proving that the latest advancements in MILP solver implementation, multi-core machines, and our in-house MILP compiler optimizations make it possible to use MILP for *online* cluster scheduling at single-digit second granularity.

Chapter 9

Discussion and Future Work

9.1 Lessons Learned

This section lays out a number of high-level takeaways that arise from this dissertation work and have broader implications.

First, we successfully demonstrate and have received anecdotal confirmation from industry researchers that, contrary to prior intuitions, it is possible and practical to use traditionally heavy-weight machinery of optimization engines for sizable clusters to schedule jobs at a single-digit second scheduling granularity. Doing so is useful for jobs that are tens of seconds or minutes in estimated duration. Mixed Integer Linear Programming (MILP) problems, in particular, are often NP-Hard. As a result, the use of MILP formulation for scheduling was largely reserved for offline resource allocation in the past. Advances in multicore technology, larger RAM, and better scalability of MILP optimization engines, such as the IBM CPLEX solver we use, coupled with optimization techniques we develop (§5.2.6) have created a possibility for online schedulers to leverage MILP as well.

Second, we find that our scheduler can be surprisingly robust to runtime mis-estimation. Foremost, this stems from the scheduler's cyclical re-evaluation of the optimal mapping of jobs to cluster resources. Continuous re-evaluation helps adapt to unpredictable cluster events—both

related to hardware and software—and take corrective action that maintains a global schedule based on the latest available cluster state snapshot and formed projections.

Third, heterogeneity in both hardware and software causes scheduling to lose its desired properties of commutativity and associativity. Indeed it can be shown under simplifying assumptions that scheduling unconstrained jobs with known runtime estimates and resource quantity demands is (a) commutative and (b) associative. Any order of placement will yield the same result in terms of cluster space-time allocation. Only capacity supply and demand need to be considered. Heterogeneity breaks these properties and engenders and exponentially large space of placement options. It is no longer about aggregate capacity allocation. Even the simplest forms of heterogeneity (§2.1) create the need to differentiate between capacity drawn from resource pools of different types. More complex forms of heterogeneity (e.g., combinatorial heterogeneity) differentiate between different subsets of resources, even when they are of the same type. The takeaway is that heterogeneity (as defined in §2.1) extends beyond capacity differences and resource attribute differences. The ambiguity of the "heterogeneity" term (which we clarify in §2.1) may have masked the extent of the challenge it poses for production clusters—a challenge for which prior scheduling solutions were insufficient.

Fourth, as was shown in §7.4.2, TetriSched's ability to perform "global scheduling" is essential for heterogeneous cluster scheduling with soft constraints. We expressly distinguish TetriSched's ability to perform orderless scheduling from other scheduling techniques that perform batching and re-ordering of jobs in a pending queue. When dealing with heterogeneous jobs, specifically with soft constraints, it can be shown that any order of jobs may produce suboptimal results. Only *simultaneous* consideration of all pending jobs' constraints guarantees optimal arbitration of their constraints. Doing so also obviates the need to order jobs to be considered for placement.

9.2 When is impact greatest?

The impact of our work for a given environment is governed by a number of factors. First, the extent and the complexity of heterogeneity plays a key role. In purely homogeneous environments, existing solutions will be as effective. In heterogeneous environments however, more benefit from TetriSched is derived with increased percentage of cluster space-time demand that comes with placement constraints. As expected, the benefit for hybrid mixes of constrained and unconstrained workloads is proportional to the fraction of total demand that explicitly differentiates between different types and sets of resources.

Second, jobs with soft constraints, namely, those that have a number of admissible placement options of varying degree of fitness, benefit from STRL's ability to capture those constraints declaratively. We quantify the benefit extracted from TetriSched's soft constraint awareness in §7.1. In cluster environments where jobs have exclusively hard constraints or no constraints whatsoever, the expressive power of STRL would be underutilized and reduced to the use of language primitives (§3.3.1) alone. Jobs with soft constraints benefit from the full spectrum of STRL's expressivity by using the language operators (§3.3.2) as well.

Third, even in environments with heterogeneity and soft constraints, the relative merit associated with different placement options plays an important role. If the relative merit difference is insignificant (see §7.1.2), the difference between scheduler outcomes also becomes insignificant and approaches the behavior of simpler scheduling policies (e.g., `None`). Indeed, in Fig. 7.4, both the aggregate value and the mean response time for TetriSched approaches that of a much simpler `None` scheduling policy for minimal slowdown factors on the x-axis. The same effect can be observed by looking at Fig. 7.6. As the slowdown factor increases, so does the gap between the `None` policy and TetriSched. At the other extreme of the relative merit continuum, excessively large benefit associated with certain placement options can effectively be treated as a hard constraint. TetriSched's benefit of comprehending soft constraints is the highest when the relative merit of alternative placement options are between 1.5x and 5x.

Fourth, as demonstrated in Fig. 7.6, workloads with increased inter-arrival burstiness benefit from TetriSched's ability to plan ahead the most. The reason for this is that burstier workloads can create transient overload for all or a part of cluster resources (e.g., machines with an FPGA could be experiencing the highest load when web search index is recomputed). These transient overloads could intuitively be visualized as ripples in the cluster resource space-time. TetriSched can "smooth out" these ripples in space by letting the excess load "spill over" onto suboptimal placement options. TetriSched can further smooth out the transient overload in time by considering a fraction of the pending load for deferral, leveraging jobs' declaratively specified flexibility in the time dimension.

Fifth, not surprisingly, load is a significant contributing factor affecting the utility of proposed mechanisms. As discussed in §7.1 (Fig. 7.2), lower load reduces the need for more advanced scheduling solutions simply because the cluster is over-provisioned enough to accommodate imperfect allocation decisions. Specifically, we found that with $\rho < 0.5$ the difference between TetriSched and `Hard` is insignificant. Increasing ρ, however, quickly creates a significant separation between them. We study the effect of partial partition load further by looking at scheduler performance as we offer progressively unbalanced workload in Fig. 7.2. This confirms our intuition that, even when the aggregate load is low, finer granularity load on individual resource partitions still plays an important role, as some partitions may dynamically become more contended than others. Such unbalanced workloads (relative to the composition of the cluster) significantly benefit from the ability of TetriSched to smooth out transient overloads, widening the scope of TetriSched's effectiveness.

There are limitations to what STRL can express, and we view this as a great opportunity for future work (§9.3). STRL expressions, by design, operate on resource subsets. Recall that, mathematically, we model a soft constraint as a function that maps arbitrary resource subsets to $\mathbb{R}$. Resource subsets are declaratively specified using any one of the family of resource attribute description languages. As such, resource subsets can be specified only as a function of

resource attributes. They cannot be specified based on the property of pending jobs. Once a job is scheduled, its unique identifier may serve as an attribute of the set of resources occupied. Before that, however, any preferences that wish to express a relationship between pending jobs cannot be specified and serviced in a single cycle. Nevertheless, the expressiveness of STRL and its implementation in TetriSched is general enough to apply in a variety of resource management contexts, including non-systems scenarios, such as serving bids for a heterogeneous fleet of cars or drones.

9.3 Future Work

9.3.1 Cloud Federation and Brokerage

A growing number of Infrastructure-as-a-Service (IaaS) providers competing on the resource spot market presents an interesting opportunity to leverage all of them simultaneously, serving as a proxy to the customers who just want to run their jobs. The extent of heterogeneity in the offerings of each such provider(e.g., Amazon, Google, and Microsoft as the biggest players) necessitates a principled general way of capturing this heterogeneity in a unified fashion across providers and scheduling jobs against a set of thus abstracted resources. TetriSched's approach is well-positioned as a significant step in this direction with its ability to quantify and express a large number of placement options succinctly with STRL. Some open challenges include handling dynamic changes in the value or cost of those placement options, particularly in the context of spot market price variation.

9.3.2 Probabilistic Scheduling

There are several sources of non-determinism in our model. First, at scale, machines have a greater probability of downtime, making equivalence sets more fluid and capacity constraints more probabilistic, especially as we consider jobs for deferral further into the future. Second,

job runtimes themselves are subject to mis-estimation, even when we run exactly the same job on exactly the same set of resources with no interference. This calls for a probability-centric formulation of the scheduling problem, where value is calculated probabilistically instead of deterministically. We've introduced one ad hoc approach aimed in this direction with our mechanism for over-estimate handling. Namely, instead of a sharp drop-off in a job's utility function at the deadline point, we simply lower the expected value of the job linearly with time past the deadline. But, casting the value calculation in a more explicitly probabilistic framework would make it possible to rest both resource unavailability and job runtime mis-estimation handling onto the same mathematical scaffolding of probabilistic value estimation.

9.3.3 Increased plan-ahead window

Currently, TetriSched must limit the plan-ahead window to avoid excessive MILP solver time (or solution quality degradation due to an early solver interrupt). But, it should be possible to increase the window of resource space-time over which we bin-pack cluster jobs. Doing so would require a mechanism for more carefully identifying specific points of interest in time, when the scheduler should trigger its core STRL aggregation, MILP formulation, and MILP solution engine. Intuitively, if the state of the cluster is identical between two points in time, there's no need to recompute the schedule. It's an extreme example that rarely happens in a large cluster. However, focusing on the delta change in the cluster between two time points could yield a much smaller MILP formulation and scale better with faster achievable scheduling latency.

9.3.4 Managing heterogeneous fleet of resources

More generally, the conceptual contributions of this work lend themselves very nicely to any context where a set of heterogeneous resources must be allocated, and consumers have explicit or implicit preferences over those resources. In the new era of autonomous vehicles and drones, their management and allocation is reminiscent of the cluster resource management context we

focus on in this dissertation. Furthermore, the time granularity of fleet allocation decisions is such that it can easily tolerate several seconds of solver latency spent arriving at an optimal allocation solution.

9.3.5 Fairness in a heterogeneous context

There are two problems with fairness as it's commonly defined. First, it focuses on counting resource consumption, instead of the higher-level objectives of resource consumers and their satisfaction. Proportional allocation of resources may not, in general, lead to proportionally satisfied consumers. Indeed, intuitively, given a mix of coconut and bananas, fairness will not be achieved by proportionally dividing each pool of fruit, if some people are allergic to coconut, some prefer bananas, and some are indifferent, as long as they get some fruit. It would be achieved only if (a) all fruit were the same or (b) all persons were indifferent. Defining fairness calls for a more nuanced modeling approach that captures and normalizes this heterogeneity. We believe that defining fairness for environments with soft and combinatorial constraints is an interesting question for future work that uses utility to quantify fairness across diverse users with declaratively specified preference structures over a heterogeneous pool of resources.

Second, it's typically defined as an instantaneous property of resource distribution, lacking any notion of time. The following example illustrates how it creates a problem. Two jobs ask for 3 nodes each for 2 time units on a 4 node cluster with a deadline in 4 time units. Clearly, both jobs can meet their deadline if serialized. Further, any sequential execution of these jobs is intuitively *fair*, since both get the same amount of cluster *space-time*. However, viewed through the lens of instantaneous fairness, such allocations can be perceived as unfair, since it dictates that resources be shared equally when both jobs have pending demand. Honoring that would result in both jobs sharing the cluster simultaneously. If such coscheduling causes interference (likely), jobs may miss their deadlines, violating their higher-level objectives. Indeed, instantaneous fairness can hurt performance unnecessarily as it tends to maximize the number of tenants simultane-

ously sharing resources and, therefore, interference. Extending the definition of fairness into the 2D cluster *space-time*—the notion we introduce in this dissertation—is a generalization of the traditional instantaneous approach to quantifying fairness.

Chapter 10

Conclusion

This dissertation validates the hypothesis that it is possible and beneficial to explicitly express (i.e., declare) jobs' spatiotemporal resource placement preferences in cluster execution environments that exhibit complex heterogeneity and dynamicity. The result is a general-purpose *space-time soft constraint scheduling* framework that enjoys an expanded, two degrees of freedom view of cluster space-time and a wider selection of declared allocation options with varying degrees of preference.

We address the challenge of scheduling for such heterogeneous environments directly, aiming to provide a highly general, extensible, and widely applicable framework for expressing, comprehending, and leveraging declarative space-time soft constraints. The key insight of this work is the *simultaneity* of considering and leveraging exponentially many choices in the space-time state of the cluster to choose better schedules that maximize the aggregate efficiency of pending jobs. To do so, we design a Space-Time Request Language (see STRL in §3)—the main theoretical contribution of this work—along with the language compiler that automatically translates STRL expressions to the intermediate representation in the form of a canonical Mixed Integer Linear Programming problem formulation. These two theoretical contributions form the foundation for the system we architect, called TetriSched, that instantiates a list of our conceptual contributions: (a) declarative soft constraints, (b) space-time soft constraints, (c) combinatorial

constraints, (d) orderless global scheduling, and (e) in situ preemption.

We demonstrate that these ideas can be applied in the context of cluster resource management, effectively capturing a variety of job types and their preferences specified over a range of resource types. We empirically validate that space-time soft constraints improve resource allocation decisions both in terms of a normalizing currency of utility, when jobs' objectives are not directly comparable, as well as in terms of concrete systems metrics, such as completion time SLOs, deadlines met, and best-effort latency.

TetriSched is designed from first principles to be highly general and, therefore, applicable in a variety of other resource management contexts. First, we adapt some of the ideas in collaboration with Microsoft as we design and implement a heterogeneity-aware resource reservation system called Aramid with support for ordinal placement preferences (a subset of the expressivity of soft constraints), targeting deployment in production clusters at Microsoft scale (see §8). A range of practical considerations there restricts the generality of space-time soft constraints that could be reasonably and widely deployed in a single release. A sequence of steps, forming an adoption on-ramp is, therefore, preferred, and we've taken the first significant step with Aramid. Second, TetriSched ideas found applications in the context of execution thread placement on manycore Network-on-Chip processors, such as Tilera's TilePro64 [67]. Third, space-time soft constraints may even find application outside the realm of computer systems altogether. The notion of flexible preferences in terms of types of resources, readily available resource usage duration estimate, and flexibility of resource access start time is directly applicable to fleet management, e.g., short-term car rentals, such as the (currently manual) reservation-based service offered by ZipCar today.

Bibliography

[1] Hadoop fair scheduler, 2013. http://hadoop.apache.org/docs/stable1/fair_scheduler.html. 2.2

[2] Scale-out resource management at microsoft using apache yarn, 2016. URL http://www.slideshare.net/HadoopSummit/scaleout-resource-management-at-microsoft-using-apache-yarn. 8.5

[3] Khalil Amiri, David Petrou, Gregory R. Ganger, and Garth A. Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, pages 25–25, Berkeley, CA, USA, 2000. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267724.1267749. 2.2

[4] Benjamin Avi-Itzhak and Hanoch Levy. On measuring fairness in queues. *Advances in Applied Probability*, 36(3):919–936, September 2004. URL http://www.jstor.org/stable/4140415. 2.3

[5] Arka A. Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. Hierarchical scheduling for diverse datacenter workloads. In *Proc. of the 4th ACM Symposium on Cloud Computing*, SOCC '13, 2013. 2.2

[6] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin. 2.4, 2.4.1

[7] Larry Brown. cuDNN v2: Higher Performance for Deep Learning on GPUs, 2015. http://goo.gl/Z4vpk2. 8.1

[8] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The case for evaluating mapreduce performance using workload suites. In *MASCOTS*, 2011. 6.2.4, 8.4.3

[9] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. In *Proceedings of the VLDB Endowment*, PVLDB, 2012. 6.2.4, 8.4.3

[10] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based Scheduling: If You'Re Late Don'T Blame Us! In

Proceedings of the ACM Symposium on Cloud Computing, SOCC, 2014. 1.1, 2, 2.4, 2.4.1, 3.4.1, 2, 5, 5.2.5, 8.2, 8.3.2, **??**

[11] Carlo Curino, Subru Krishnan, Ishai Menache, Seffi Naor, Sriram Rao, and Jonathan Yaniv. LowCost: A cost-based reservation algorithm for hadoop yarn++. Technical report, Microsoft, Inc., 2015. URL `https://issues.apache.org/jira/browse/YARN-3656`. 8.2

[12] Jun Shi Cyprien Noel and Yahoo Big ML Team Andy Feng. Large Scale Distributed Deep Learning on Hadoop Clusters, 2015. `http://goo.gl/m3pCvt`. 8.1

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL `http://doi.acm.org/10.1145/1327452.1327492`. 2.2

[14] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 499–510, Berkeley, CA, USA, 2015. USENIX Association. ISBN 978-1-931971-225. URL `http://dl.acm.org/citation.cfm?id=2813767.2813804`. 2.4.1

[15] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451125. URL `http://doi.acm.org/10.1145/2451116.2451125`. 2.1, 2.2, 3.4.1, 6.2.2

[16] Christina Delimitrou and Christos Kozyrakis. Qos-aware scheduling in heterogeneous datacenters with paragon. *ACM Trans. Comput. Syst.*, 31(4):12:1–12:34, December 2013. ISSN 0734-2071. doi: 10.1145/2556583. URL `http://doi.acm.org/10.1145/2556583`. 2.1, 2.4, 3.4.1

[17] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541941. URL `http://doi.acm.org/10.1145/2541940.2541941`. 2.2, 2.4

[18] Ankush Desai, Kaushik Rajan, and Kapil Vaswani. Critical path based performance models for distributed queries. In *Microsoft Tech-Report: MSR-TR-2012-121*, 2012. 2.4, 2.4.1, 3.4.1

[19] Dror G Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling—a status report. In *Job Scheduling Strategies for Parallel Processing*, pages 1–16. Springer, 2004. 2.4

[20] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proc. of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 99–112, 2012. ISBN 978-1-4503-

1223-3. doi: 10.1145/2168836.2168847. URL http://doi.acm.org/10.1145/2168836.2168847. 1.1, 2.3, 2.4, 2.4.1, 3.4.1, 6.1.3

[21] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI, 2011. 2

[22] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 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. 2.2, 2.3

[23] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 365–378, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. doi: 10.1145/2465351.2465387. URL http://doi.acm.org/10.1145/2465351.2465387. 2.3

[24] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. PRObE: A thousand-node experimental cluster for computer systems research. *USENIX ;login:*, 38(3), June 2013. URL https://www.usenix.org/publications/login/june-2013-volume-38-number-3/probe-thousand-node-experimental-cluster-computer. (document), 6.2.1

[25] M. Guevara, B. Lubin, and B.C. Lee. Strategies for anticipating risk in heterogeneous system design. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 154–164, Feb 2014. doi: 10.1109/HPCA.2014.6835926. 2.2

[26] Marisabel Guevara, Benjamin Lubin, and Benjamin C. Lee. Market mechanisms for managing datacenters with heterogeneous microarchitectures. *ACM Trans. Comput. Syst.*, 32 (1):3:1–3:31, February 2014. ISSN 0734-2071. doi: 10.1145/2541258. URL http://doi.acm.org/10.1145/2541258. 2.2

[27] Stefan Hadjis, Firas Abuzaid, Ce Zhang, and Christopher Ré. Caffe con troll: Shallow ideas to speed up deep learning. In *Proceedings of the Fourth Workshop on Data Analytics in the Cloud*, DanaC'15, pages 2:1–2:4, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3724-3. doi: 10.1145/2799562.2799641. URL http://doi.acm.org/10.1145/2799562.2799641. 8.1

[28] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems (TOCS)*, 15(3):253–285, August 1997. ISSN 0734-2071. doi: 10.1145/263326.263344. URL http://doi.acm.org/10.1145/263326.263344. 2.4.1

[29] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R.H. Katz, S. Shenker, and I. Stoica. 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. 2.1, 2.2

[30] David Irwin, Jeff Chase, Laura Grit, and Aydan Yumerefendi. Self-recharging virtual currency. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Economics of Peer-to-peer Systems*, P2PECON '05, pages 93–98, New York, NY, USA, 2005. ACM. ISBN 1-59593-026-4. doi: 10.1145/1080192.1080194. URL http://doi.acm.org/10.1145/1080192.1080194. 2.3

[31] David Irwin, Jeff Chase, Laura Grit, and Aydan Yumerefendi. Self-recharging virtual currency. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Economics of Peer-to-peer Systems*, P2PECON '05, pages 93–98, New York, NY, USA, 2005. ACM. ISBN 1-59593-026-4. doi: 10.1145/1080192.1080194. URL http://doi.acm.org/10.1145/1080192.1080194. 2.3

[32] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceeding of the ACM Symposium on Operating Systems Principles*, SOSP, 2009. 2.2

[33] Mohammad Islam, Angelo K Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. Oozie: Towards a Scalable Workflow Management System for Hadoop. In *SWEET Workshop*, 2012. 2.4

[34] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 407–420, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3542-3. doi: 10.1145/2785956.2787488. URL http://doi.acm.org/10.1145/2785956.2787488. 2.4.1

[35] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014. 1, 2.1

[36] Terence Kelly. Utility-directed allocation. Technical Report HPL-2003-115, Internet Systems and Storage Laboratory, HP Labs, June 2003. URL http://www.hpl.hp.com/techreports/2003/HPL-2003-115.html. 2.3

[37] Terence Kelly. Combinatorial auctions and knapsack problems. In *Proc. of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '04, pages 1280–1281, 2004. ISBN 1-58113-864-4. doi: 10.1109/AAMAS.2004.84. URL http://dx.doi.org/10.1109/AAMAS.2004.84. 2.3

[38] S. Krishnaswamy, S. W. Loke, and A. Zaslavsky. Estimating computation times of data-intensive applications. *IEEE Distributed Systems Online*, 5(4), April 2004. ISSN 1541-4922. doi: 10.1109/MDSO.2004.1301253. 2.4.1

[39] Kevin Lai, Lars Rasmusson, Eytan Adar, Li Zhang, and Bernardo A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, 1(3):169–182, August 2005. ISSN 1574-1702. URL http://dl.acm.org/citation.cfm?id=1233813.1233816. 2.3

[40] Cynthia B. Lee and Allan E. Snavely. Precise and realistic utility functions for user-centric performance analysis of schedulers. In *Proc. of the 16th international symposium on High*

performance distributed computing, HPDC '07, pages 107–116. ACM, 2007. ISBN 978-1-59593-673-8. doi: 10.1145/1272366.1272381. URL http://doi.acm.org/10.1145/1272366.1272381. 2.3

[41] Cynthia B. Lee and Allan E. Snavely. Precise and realistic utility functions for user-centric performance analysis of schedulers. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, HPDC '07, pages 107–116, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-673-8. doi: 10.1145/1272366.1272381. URL http://doi.acm.org/10.1145/1272366.1272381. 2.3, 2.3

[42] Cynthia Bailey Lee and Allan Snavely. On the user–scheduler dialogue: Studies of user-provided runtime estimates and utility functions. *International Journal of High Performance Computing Applications*, 20(4):495–506, 2006. URL http://hpc.sagepub.com/content/20/4/495.abstract. 2.3

[43] David A Lifka. The ANL/IBM SP scheduling system. In *Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer, 1995. 2.4

[44] J. Mars, L. Tang, and R. Hundt. Heterogeneity in 'homogeneous' ; warehouse-scale computers: A performance opportunity. *IEEE Computer Architecture Letters*, 10(2):29–32, July 2011. ISSN 1556-6056. doi: 10.1109/L-CA.2011.14. 6.2.2

[45] J. Mars, Lingjia Tang, and R. Hundt. Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity. *Computer Architecture Letters*, 10(2):29–32, July 2011. ISSN 1556-6056. doi: 10.1109/L-CA.2011.14. 2.2

[46] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 619–630, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. doi: 10.1145/2485922.2485975. URL http://doi.acm.org/10.1145/2485922.2485975. 2.2

[47] Kristi Morton, Abram Friesen, Magdalena Balazinska, and Dan Grossman. Estimating the progress of mapreduce pipelines. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 681–684. IEEE, 2010. 2.4.1

[48] I. A. Moschakis and H. D. Karatza. Performance and cost evaluation of gang scheduling in a cloud computing system with job migrations and starvation handling. In *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pages 418–423, June 2011. doi: 10.1109/ISCC.2011.5983873. 2.4

[49] Ripal Nathuji, Canturk Isci, and Eugene Gorbatov. Exploiting platform heterogeneity for power efficient data centers. In *Fourth International Conference on Autonomic Computing (ICAC)*. IEEE, 2007. 2.2

[50] John K Ousterhout. Scheduling techniques for concurrent systems. In *International Conference on Distributed Computing Systems (ICDCS)*, volume 82, pages 22–30, 1982. 2.4

[51] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM. ISBN

978-1-4503-2388-8. doi: 10.1145/2517349.2522716. URL http://doi.acm.org/10.1145/2517349.2522716. 2.2

[52] Florentina I. Popovici and John Wilkes. Profitable services in an uncertain world. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 36–, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 1-59593-061-2. doi: 10.1109/SC.2005.58. URL http://dx.doi.org/10.1109/SC.2005.58. 2.3, 2.3

[53] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-4394-4. URL http://dl.acm.org/citation.cfm?id=2665671.2665678. 1

[54] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jordan Gray, et al. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceedings of the Annual International Symposium on Computer Architecture*, ISCA, 2014. 8.1, 1, 8.4.3

[55] Rajesh Raman, Miron Livny, and Marvin Solomon. 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. 2.2, 3.2

[56] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. of the 3nd ACM Symposium on Cloud Computing*, SOCC '12, 2012. 2.1, 2.4.1, 3.1.1, 5.2.2, 6.2.2

[57] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Towards understanding heterogeneous clouds at scale: Google trace analysis. Technical Report ISTC-CC-TR-12-101, Intel Science and Technology Center for Cloud Computing, Apr 2012. URL http://www.istc-cc.cmu.edu/publications/papers/ISTC-CC-TR-12-101_abs.shtml. 2.1, 3.1.1, 5.2.2

[58] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *ACM Eurosys Conference*, 2013. 2.2

[59] Bikash Sharma, Victor Chudnovsky, Joseph L. Hellerstein, Rasekh Rifaat, and Chita R. Das. Modeling and synthesizing task placement constraints in Google compute clusters. In *Proc. of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 3:1–3:14. ACM, 2011. ISBN 978-1-4503-0976-9. doi: http://doi.acm.org/10.1145/2038916.2038919. URL http://doi.acm.org/10.1145/2038916.2038919. 1.1, 2.1, 2.2, 3.1.1, 3.2, 6.2.2

[60] Piyush Shivam, Shivnath Babu, and Jeff Chase. Active and accelerated learning of cost models for optimizing scientific applications. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 535–546. VLDB Endowment, 2006. URL http://dl.acm.org/citation.cfm?id=1182635.11641734. 3.4.1

[61] Ozan Sonmez, Nezih Yigitbasi, Alexandru Iosup, and Dick Epema. Trace-based evaluation of job runtime and queue wait time predictions in grids. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, pages 111–120, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-587-1. doi: 10.1145/1551609.1551632. URL http://doi.acm.org/10.1145/1551609.1551632. 2.4.1

[62] Ion Stoica, Hussein Abdel-wahab, and Alex Pothen. A microeconomic scheduler for parallel computers. In *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 122–135. Springer-Verlag, 1994. 2.3

[63] Roshan Sumbaly, Jay Kreps, and Sam Shah. The Big Data Ecosystem at LinkedIn. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2013. 2.4

[64] The Apache Software Foundation. GridMix, 2015. http://hadoop.apache.org/docs/current/hadoop-gridmix/GridMix.html. 8.4.1

[65] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803, June 2007. ISSN 1045-9219. doi: 10.1109/TPDS.2007.70606. 2.4

[66] Alexey Tumanov, James Cipar, Michael A. Kozuch, and Gregory R. Ganger. alsched: algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proc. of the 3nd ACM Symposium on Cloud Computing*, SOCC '12, 2012. 1.1, 2.1, 2.4, 5.2.1, 7.4.2

[67] Alexey Tumanov, Joshua Wise, Onur Mutlu, and Gregory R. Ganger. Asymmetry-aware execution placement on manycore chips. In *Proc. of the 3rd Workshop on Systems for Future Multicore Architectures (SFMA'13)*, SFMA '13, 2013. 10

[68] Alexey Tumanov, Timothy Zhu, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Space-time scheduling for heterogeneous datacenters. Technical Report CMU-PDL-13-112, Carnegie Mellon University, Nov 2013. URL http://www.pdl.cmu.edu/PDL-FTP/CloudComputing/CMU-PDL-13-112_abs.shtml. 3.2, 4, 5.2, 7.4.3

[69] Alexey Tumanov, Timothy Zhu, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Tetrisched: Space-time scheduling for heterogeneous datacenters. Technical report, Citeseer, 2013. 2.4

[70] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 35:1–35:16, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901355. URL http://doi.acm.

org/10.1145/2901318.2901355. 2.4, 5.2, 5.2.5

[71] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, , Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *Proc. of the 4th ACM Symposium on Cloud Computing*, SOCC '13, 2013. 2.2, 2.4, 5.3, 8.2, 2, 8.4.1, **??**, **??**

[72] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. The power of choice in data-aware cluster scheduling. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 301–316, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL http://blogs.usenix.org/conference/osdi14/technical-sessions/presentation/venkataraman. 1.1, 2.1

[73] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J Franklin, and Ion Stoica. The power of choice in data-aware cluster scheduling. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 301–316. USENIX Association, 2014. 2.2

[74] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016. 2.4.1

[75] S. Verboven, P. Hellinckx, F. Arickx, and J. Broeckhove. Runtime prediction based grid scheduling of parameter sweep jobs. In *Asia-Pacific Services Computing Conference, 2008. APSCC '08. IEEE*, pages 33–38, Dec 2008. doi: 10.1109/APSCC.2008.189. 2.4.1

[76] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *ICAC '11*, 2011. 1.1, 2.4, 2.4.1, 3.4.1

[77] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741964. URL http://doi.acm.org/10.1145/2741948.2741964. 2.2, 2.4.2, 3

[78] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267638.1267639. 2.2

[79] John Wilkes. Utility functions, prices, and negotiation. Technical Report HPL-2008-81, HP Labs, July 2008. URL http://www.hpl.hp.com/techreports/2008/HPL-2008-81.pdf. 2.3

[80] Timothy Wood, Prashant J Shenoy, Arun Venkataramani, and Mazin S Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, volume 7, pages 17–17, 2007. 2.4.2

[81] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-577-2. doi: 10.1145/1755913.1755940. URL `http://doi.acm.org/10.1145/1755913.1755940`. 5.2.1

[82] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems (Eurosys)*, pages 265–278. ACM, 2010. 2.4

[83] Yanyong Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):236–247, March 2003. ISSN 1045-9219. doi: 10.1109/TPDS.2003.1189582. 2.4