Tetrisched: Space-Time Scheduling for Heterogeneous Datacenters

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

CMU-PDL-13-112
December 2013

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Tetrisched is a new scheduler that explicitly considers both job-specific preferences and estimated job runtimes in its allocation of resources. Combined, this information allows tetrisched to provide higher overall value to complex application mixes consolidated on heterogeneous collections of machines. Job-specific preferences, provided by tenants in the form of composable utility functions, allow tetrisched to understand which resources are preferred, and by how much, over other acceptable options. Estimated job runtimes allow tetrisched to plan ahead in deciding whether to wait for a busy preferred resource to become free or to assign a less preferred resource. Tetrisched translates this information, which can be provided automatically by middleware (our wizard) that understands the right SLOs, runtime estimates, and budgets, into a MILP problem that it solves to maximize overall utility. Experiments with a variety of job type mixes, workload intensities, degrees of burstiness, preference strengths, and input inaccuracies show that tetrisched consistently provides significantly better schedules than alternative approaches.

Acknowledgements: We thank the member companies of the PDL Consortium (Actifio, APC, EMC, Emulex, Facebook, Fusion-IO, Google, HP, Hitachi, Huawei, Intel, Microsoft, NEC Labs, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, 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 an NSERC Postgraduate Fellowship, and by National Science Foundation under awards CSR-1116282, 0946825 and CNS-1042537, CNS-1042543 (PRObE).

Keywords: cluster scheduling, cloud systems

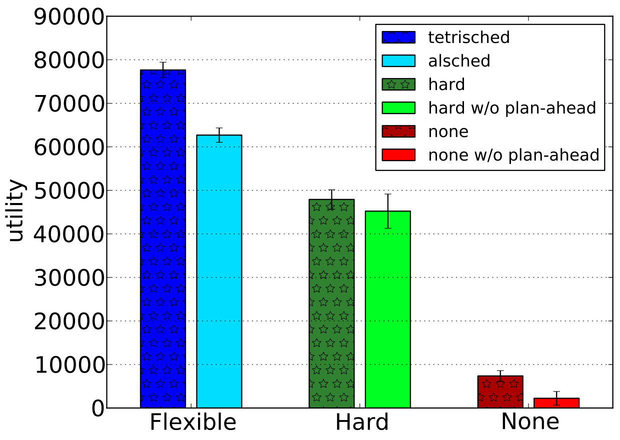


Figure 1: Better guidance leads to better scheduling. The three bar pairs correspond to schedulers that ignore constraints (`None`), that consider only hard constraints (`Hard`), and that consider soft constraints as well (`Flexible`). In each pair, the left bar exploits runtime estimates to plan ahead, 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 $C_A^2 = 8$.

1 Introduction

Datacenters increasingly use a heterogeneous collection of machines with different capabilities (e.g., memory capacity, GPU accelerator) to execute a heterogeneous collection of workloads (e.g., long-running services, batch data analytics, interactive development/test) [21, 19, 23]. To maximize resource efficiency and utilization, the machines are often aggregated into a resource pool onto which the workloads are consolidated. A cluster scheduler is then tasked with assigning resources to workloads.

The challenge is to assign the right resources to each, given varied characteristics and metrics of goodness. One job might be concerned about completion time and run fastest on a machine with a GPU. Another job might be concerned with long-term availability, which is enhanced by running its constituent processes (tasks) on machines attached to separate power distribution units. When machines are homogeneous, such concerns can often be ignored as all choices are equal. When workloads are homogeneous, understanding of their concerns can be hard-coded into the scheduler (e.g., locality awareness in Hadoop [1]). But, neither workloads nor resources are homogeneous in modern datacenters.

To achieve the promised efficiency benefits of consolidation, resource consumers must somehow communicate their specific needs/concerns so that the scheduler can make informed decisions. One approach is to associate some number of *hard constraints* with each resource request, based on some predetermined machine attribute schema, to identify the subset of machines that are suitable [22, 24]. But, this approach ignores an important issue: in many cases, desired machine characteristics provide benefit but are not mandatory. For example, running a new task on the same machine as another with which it communicates frequently can improve performance, but failing to do so does not prevent the task's execution—it just makes it somewhat less efficient. A scheduler that understands the quantitative tradeoffs involved with such *soft constraints* should be able to make better decisions [26].

Fig. 1 illustrates the benefit of schedulers that have and exploit better information about each job's concerns and needs. Following best practices of economic theory, workloads' distinct concerns (e.g., availabilities, runtimes, response times) are translated to a common metric called *utility* so that tradeoffs between them can be quantified—higher utility is better. As should be expected, the more information about significant needs/concerns used by the scheduler, the higher the utility. In this case, the most aware scheduler

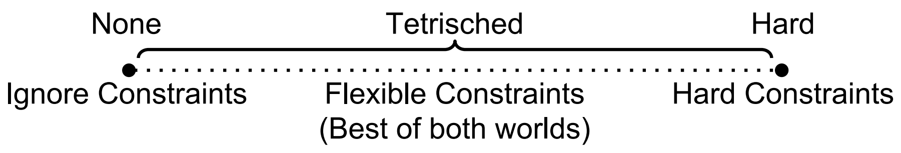


Figure 2: Tetrisched fills the constraint handling gap. Most schedulers take an all-or-nothing approach, either treating all constraints as strict requirements or completely ignoring them. Tetrisched recognizes that tradeoffs are inherent in user preferences, providing a flexible constraint scheme that encodes resource preferences and their relative utility.

(`tetrisched`) provides 58% and 14× higher utility, respectively, than those that consider all constraints to be hard or that ignore them entirely.

This paper introduces tetrisched (see Fig. 2), a scheduler that accepts resource requests in the form of utility functions. These utility functions use an algebraic language to describe the utility associated with one or more spatial (which machines) and temporal outcomes, quantifying the relative value of each acceptable allocation. Tetrisched translates the collection of utility functions into a Mixed Integer Linear Program (MILP) problem and solves it to plan a schedule that optimizes overall utility. Extensive simulation study of a heterogeneous 1000-node cluster shows that tetrisched consistently and significantly outperforms less-informed approaches to scheduling, across different workload mixes, workload intensities, degrees of burstiness, and preference strengths. We also show that its decisions are robust to user mis-estimation of job size.

This paper makes several important contributions over prior work, including our recent alsched position paper [26]. First, it exposes the importance of making schedulers aware of workload-specific preferences regarding both time and space. As illustrated in the Fig. 1 example, neither alone is sufficient, with tetrisched outperforming the best space-only (alsched) and time-enhanced non-soft (Hard) options by 33% and 58%,[1] respectively. Second, it describes a language for crisply specifying space-time concerns as utility functions. Third, it describes a user objective wizard (middleware) that automatically translates completion-time SLOs, budget constraints, and runtime estimates into these utility functions. Fourth, it provides extensive evaluation of tetrisched's behavior along many axes, explaining how different utility functions drive the choices that it makes and how well it satisfies complex mixes of workload concerns.

2 Context and Motivation

Scheduling multiple types of jobs among heterogeneous machines is complex. Consider the example in Fig. 3, which shows five space-time schedules for three jobs that each want two servers but are best served by differing resource characteristics. All five schedules are viable options, and no schedule is ideal for all three jobs. Not only does some job have to wait its turn, but the Availability job cannot run concurrently with either of the other two unless some job uses non-preferred machine mixes.

Designing a scheduler to produce these optimized space-time schedules is an achievement, but if the caption's specification is the only information available to the scheduler, then the scheduler cannot determine which of these schedules is the best. Fundamentally, the scheduler needs a concise representation of user sensitivity to delay, availability, etc., as well as the importance of the job. For example, if the Availability job is insensitive to delay, then the bottom schedule will be the best choice. However, if the Availability job is much more revenue generating than the other jobs, then the middle schedules are more appealing. Tetrisched takes in such user objectives and optimizes for the best overall tradeoff. The remainder of this

[1]As we'll see in section 7, tetrisched can outperform these other options by 3x or more under certain conditions.

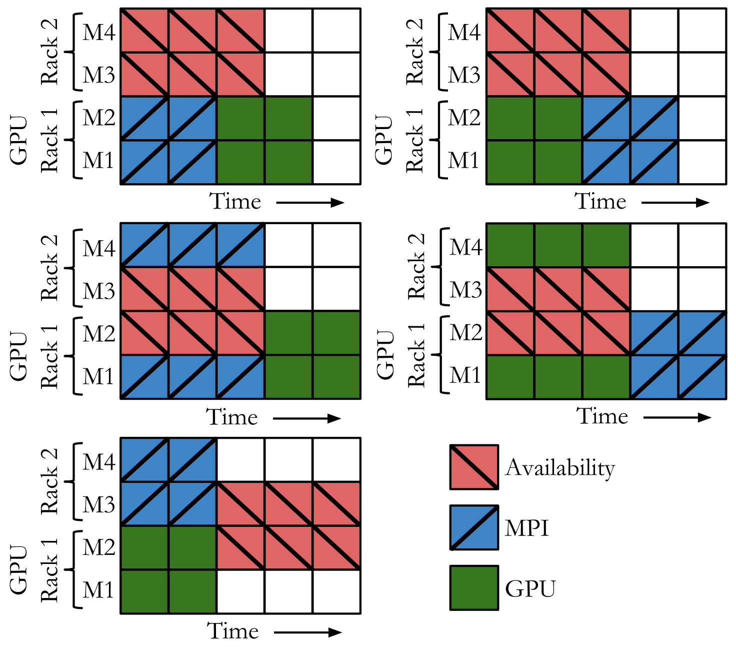


Figure 3: Five potential schedules for 3 jobs. Each grid shows one potential space-time schedule, with machines along the rows and time units along the columns. Each job requests 2 servers, and its allocation is shown by filling in the corresponding grid entries. The cluster consists of 2 racks each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

section discusses the user and provider perspectives underlying the tetrisched approach, and then describes the system usage model.

2.1 User and Provider Perspectives

User Objectives. As exemplified in Fig. 3, users may have fundamentally different objectives, workload characteristics, and incentives. Examples of user objectives may include finishing work as soon as possible for a GPU job, maximizing uptime for an ensemble of web services (Availability), or starting a long-running service as soon as possible. Specific examples we consider include:

– response time (completion time minus arrival time): users prefer to minimize the overall completion time for submitted jobs. Examples include a number of batch jobs, such as MPI-based simulations or Hadoop analytics jobs, and machine learning jobs.

– queueing delay (start time minus arrival time): preference is for minimal queueing delay. User-facing interactive applications, such as augmented reality servers, are good examples of queueing delay sensitivity.

– availability: expected availability or fault-tolerance of a given ensemble of web services.

– expected quality of service output.

Many jobs that have the last two user objectives, such as long-running services, often also have quantifiable preference for minimizing queueing delay. Full exposition of such multi-objective control is deferred to section 7.4. Similarly to Jockey [10], we use time-based utility functions depicted in Fig. 4 as one form of high-level input into tetrisched. For completion-oriented jobs, the top plateau of the utility function is most desirable as it yields the highest utility. As the calculated completion time falls between the Desired and Deadline points on the x-axis, utility starts to decline. After a certain point, utility may drop below zero,

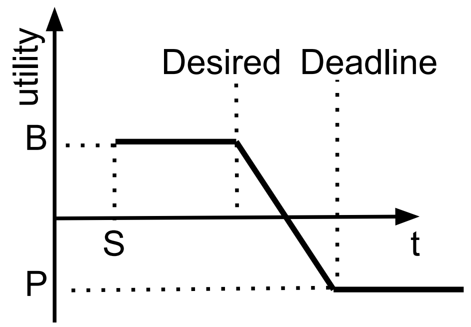


Figure 4: Users' subjective utility over time. Temporal user objectives can be modeled as a time-based user-defined utility function (uduf). S stands for earliest start time and allows support for expressing calendared jobs. Budget B is the max utility this job gets. Penalty P is a negative amount of utility accrued for failure to meet the Deadline. Utility starts to decline when the desired time objective is not reached.

inducing a utility penalty if a job wasn't scheduled in a reasonable amount of time. The same user-defined utility function ("uduf", see Fig. 4) shape can be used for specifying queueing delay preferences, with times on the x-axis representing queueing delay instead of response time.

Provider Perspective. Heterogeneity in both user objectives and increasingly diverse hardware introduces new scheduling challenges as cluster resource providers face more tradeoffs. Providers often deal with heterogeneity by trying to eliminate it or providing a mechanism to force conformance to a certain supported standard. The latter approach doesn't scale, and neither approach addresses the fundamental problem of arbitrating inherent contention for resources and making reasonable tradeoffs in a heterogeneous cluster environment. Instead, providers need a way to quantify the effects of tradeoffs and optimize their scheduling decisions to meet provider's goals. Goals can include maximizing revenue or customer-base happiness. In both cases, a common currency of compensation is imperative, prompting us to adopt utility as such a metric. Throughout this paper, we assume that providers wish to maximize the aggregate utility across all active cluster users.

Optimizing aggregate utility puts the global good above giving each individual an equal share of resources, for some definition of equal that considers heterogeneity and preferences. Some users may receive lesser service. An economic notion of fairness can be realized by charging users according to the utility indicated in their utility functions. These charges could be in dollars, in for-profit environments, or "credits" in non-profit environments. For the latter, the distribution of credits allows for prioritized notions of fairness, where users spend their limited resources according to their preferences.

Economics of user incentives. A utility-based approach should work for users, incentivizing them to "play nice", both in for-profit and non-profit environments. In both cases, flexibility in specified constraints ultimately maximizes the probability of resource allocation, especially when load is high and requests are large. Any profit margin that may exist behind user-specified utility curves is amplified by higher probabilities of resource acquisition. In both for-profit and non-profit environments, a higher probability of getting resources translates into higher volume of useful work done. That in turn either amplifies the profit margin or increases user happiness (higher utility).

We do not make any assumptions about the truthfulness of user-specified utility. Tetrisched takes this as a ground truth and allocates resources given the budget and the utility curve provided. Users will be constrained by their budget in how much they overstate subjective value of resources. Understating that value will lower their probability of being scheduled. Thus, the "invisible hand of the market" will govern the price-setting, dynamically adjusting it relative to the supply and demand for resources.

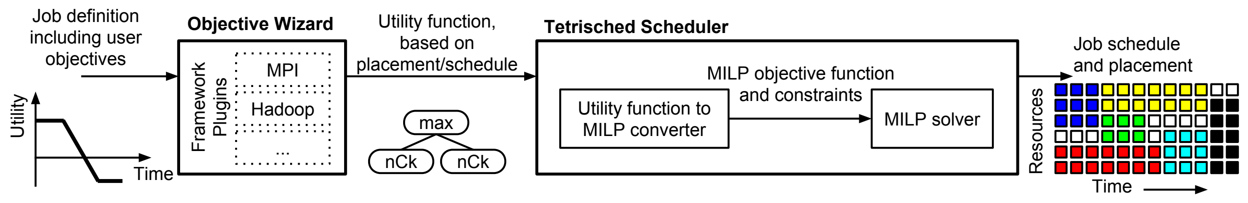

Job definition including user objectives
Utility
Time
Objective Wizard
Framework Plugins
MPI
Hadoop
...
Utility function, based on placement/schedule
max
nCk
nCk
Tetrisched Scheduler
Utility function to MILP converter
MILP objective function and constraints
MILP solver
Job schedule and placement
Resources
Time

Figure 5: System Overview. The *Objective Wizard* converts user job definitions including objective specifications into utility functions, and the *Tetrisched Scheduler* converts the utility functions into a Mixed-Integer Linear Program (MILP) representation and uses a MILP solver to generate a placement and schedule.

2.2 System Usage Model Overview

Fig. 5 illustrates the tetrisched components and how they interact. While a user could interact directly with the low-level scheduler interface, most will use a higher-level interface provided by a wizard that produces the utility functions. An objective wizard for given job types can take in job definitions that include user objectives (e.g., defined by the trapezoid vertices) and translate them to a scheduler-facing composable utility function. The latter takes the form of an algebraic expression tree that will be further discussed in Section 3.

This resource request is subsequently managed by the tetrisched scheduler that fires at specified intervals of time. At each tetrisched cycle, all outstanding resource requests are aggregated and converted into an MILP formulation. Solving it produces the job schedule that maps tasks for satisfied jobs to machines.

3 Placement Preference Specification

This section introduces the specification language for expressing user placement considerations in space and time. Three principal goals of this language are: (a) structural support for expressing *subsets* of cluster resources and quantity desired, (b) support for *grouping* subsets and avoiding redundant enumeration of combinatorial choices, and (c) support for associating *value* with a given choice of resources for a specified interval of time. The tetrisched Wizard consumes *uduf*s (Fig. 4) and produces composable scheduler expressions using the language specification we introduce here. This is accomplished with just two simple, composable *combinatorial* primitives, which can be composed with a handful of algebraic binary and unary operators—all defined in Sec. 3.3. These algebraic composable utility functions serve as input to the tetrisched scheduler core. Power users can specify them directly to the scheduler, bypassing the Wizard.

3.1 Equivalence classes and partitions

An important notion in tetrisched is that of *equivalence classes*, which are equivalent sets of machines from the perspective of a given job. For example, the GPU job in Fig. 3 is equally happy with machine M1 or M2 since both are GPU-enabled, but not as happy with machine M3. Furthermore, a GPU job is equally happy with any k machines from the set of GPU-enabled machines (e.g. M1, M2). The ability to represent sets of machines that are equivalent to a job greatly reduces the complexity of the scheduling problem because it obviates the need to enumerate all combinatorial choices of machines.

For the solver to be able to utilize equivalence classes, the machines must be considered equivalent to *all* users. Since multiple users may have overlapping equivalence classes, we need to treat each overlapping set separately. In other words, the scheduler needs to identify *partitions*, which are sets of machines that are equivalent from the perspective of *any* job currently being scheduled. Instead of statically defining partitions, we developed an algorithm to dynamically calculate the partitions based on the equivalence classes of currently queued jobs. This keeps the number of partitions to a minimum, which reduces the burden on the scheduler. The partitioning algorithm is given in Appendix A.1.

3.2 Utility Functions

Conceptually, an *algebraic utility function* defines the value of a set of machines over a range of time for a specific job. We refer to these machines over time as a space-time rectangle [2]. For example, in the bottom schedule in Fig. 3, the GPU job gets the space-time rectangle corresponding to machines M1 and M2 for times 0 and 1. Utility functions are best represented as an algebraic expression tree with operands at its leafs and operators, modifying the upward flow of utility from their children.

3.3 Language Specification

As initially introduced in [26], we have leaf primitives and non-leaf operators that act on one or more child subtrees. Tetrisched extends primitive definitions to add support for plan-ahead. Thus, tetrisched space-time primitives are defined as follows:

– $nCk(eq,k,s,dur,u)$: out of the specified equivalence class eq, choose k machines starting at start time s for duration dur to get utility u.

– $LnCk(eq,k,s,dur,u)$: out of the specified equivalence class eq, choose up to k machines starting at start time s for duration dur to get utility u. A choice of $k' < k$ returns utility $\frac{u}{k} \cdot k'$

Relative to our predecessor [26], we extend the “n Choose k” and its linear “n Choose k” counterpart by qualifying *when* k machines chosen from eq yield specified utility u. This is defined by a time interval $[s;s+dur)$. It helps to visualize the nCk building block as a function assigning scalar values to arbitrary rectangles in resource space-time. In Fig. 3, each of the (potentially non-contiguous) rectangles can be expressed with an nCk primitive (see section 3.4).

The operators are min, max, sum, barrier, and scale and are defined as follows:

– $min(t_1,...,t_n)$: returns the minimum resulting utility of the specified set of subtrees $t_1,...,t_n$. This forces a certain minimum utility across all subexpressions of the min operator and, therefore, semantically behaves like an AND.

– $max(t_1,...,t_n)$: returns the maximum resulting utility of the specified set of subtrees $t_1,...,t_n$. This picks a subexpression of maximum utility and, therefore, semantically behaves like an OR.

– $sum(t_1,...,t_n)$: returns the sum of utility values evaluated across all specified subtrees $t_1,...,t_n$.

– $scale(t,s)$: unary scaling operator scaling the utility value of subtree t by scaling factor s.

– $barrier(t,u)$: returns utility u if the utility of subtree t is $\geq u$, or 0 otherwise.

3.4 Examples

To demonstrate how the language describes user placement preferences, we show an example for the GPU job in Fig. 3. For each start time $s \in [S, Deadline)$ (see Fig. 4), we have the following two choices:

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

where $u_G(t)$ is the user-defined utility function mapping absolute completion times to utility based on the uduf shown in Fig. 4. The first choice represents getting GPU-enabled servers, which will complete in 2 time units, and the second choice represents running anywhere, but at a slower run time of 3 time units. To

[2]As can be seen in Fig. 3 “rectangles” can, of course, be non-contiguous.

combine these choices, we use the *max* operator, resulting in the following expression:

$$\begin{aligned} max(&\forall s \in [S; Deadline)\ nCk(\{M1, M2\}, k = 2, s, dur = 2, u_G(s+2)), \\ &\forall s \in [S; Deadline)\ nCk(\{M1, M2, M3, M4\}, k = 2, s, dur = 3, u_G(s+3))) \end{aligned}$$

Tetrisched combines such expressions for all jobs pending placement with a top-level *sum* operator to form the global optimization expression.

4 MILP Formulation

Tetrisched's optimization problem can be represented as a Mixed Integer Linear Program (MILP). In this formulation, we maximize a linear objective function over a set of binary, integer, and continuous variables subject to a set of linear optimization constraints. At first, it seemed that our combinatorial placement constraints were too complex to be represented as linear optimization constraints. We later realized that branches in the tetrisched expressions correspond to placement choices or placement constituents, which can be formulated in an MILP using binary indicator decision variables. For each branch, we assign an indicator variable that represents if the branch was chosen. The utility for each branch appears in the objective function multiplied by its indicator variable. Thus, the indicator variable is able to convey those branches from which we extract utility. To ensure the scheduler stays within resource limits, we also have integer partition variables that represent the quantity of resources consumed by nCk and LnCk leaf nodes. These are used in two types of optimization constraints: *supply* constraints and *demand* constraints. Supply constraints limit the scheduler to only use the available resources. Demand constraints indicate the resource requirement for satisfying the nCk and LnCk leaf nodes. Using these ideas, we recursively generate an MILP problem for the tetrisched expression corresponding to the sum of all queued jobs. The MILP generation algorithm is given in Appendix A.2.

5 System Implementation

Fig. 5 shows the major components of tetrisched. Users typically[3] submit job definitions and user objectives to the wizard. The wizard builds a tetrisched utility function that defines the user's value for space-time rectangles. This is passed to the tetrisched scheduler, which queues jobs and computes job schedules. At every scheduling cycle, the scheduler produces a job schedule for the cluster manager, which handles the launching and monitoring of jobs. We now expand in more detail on three important features: the scheduler (Sec. 5.1), plan-ahead (Sec. 5.2), and the wizard (Sec. 5.3).

5.1 Scheduler

The scheduler's primary function is to produce space-time schedules for the currently queued jobs. On each scheduling cycle, the scheduler aggregates job utility expressions across all pending jobs under a single *sum* expression[4]. This global *sum* expression is then converted into an MILP problem and fed to the solver. We use IBM CPLEX [6] for the purposes of this paper. Given the complexity and size of MILP models generated from scheduler expressions, the solver is configured to return "good enough" solutions within 10% of the optimal solution. Additionally, we cache solver results to serve as a feasible starting solution for subsequent solver invocations. We found such result reuse to be quite effective for reducing solver execution time.

[3]Power users can bypass the wizard and directly submit tetrisched utility functions to the scheduler.

[4]Other aggregation operators are possible, enabling different optimization goals, including fairness.

5.2 Plan-ahead

An important aspect of tetrisched is considering future placements for multiple jobs with flexible constraints. This allows tetrisched a wider range of options for where and when to schedule jobs in cluster space-time. It is particularly important for the scheduler to know whether it should wait for preferred resources. In Sec. 7.3, we will see that plan-ahead can lead to factor of 3 improvements. However, planning very far into the future is computationally expensive and provides diminishing returns. Since we do not have oracular knowledge, large planning horizons could even produce worse results. Thus, we limit how far tetrisched plans into the future, which we define as the plan-ahead parameter. For example, the plan-ahead parameter in Fig. 3 is 5 time units.

5.3 Wizard

We use the term "wizard" to refer to a tool that translates specified user desires into utility functions. To allow comparisons between jobs of different types, a user must specify a budget associated with each job. This can be based on priority, job size, or some other user perceived valuation. Users can also specify a penalty for dropping jobs. This allows providers the ability to form SLAs with a penalty for failing to run jobs by the specified deadline. Users specify their sensitivity to delay by providing deadlines and desired completion times. Fig. 4 shows how utility is affected as a function of response time. For power users, the wizard is also generic enough to utilize any other function relating utility to completion time. Lastly, the wizard takes an estimate of a job's duration[5]. In Sec. 7.5, we show the robustness of tetrisched to inaccuracies in job duration specification.

We now present five prototypical job types currently supported by our wizard: Unconstrained, GPU, MPI, HA, and HDFS.

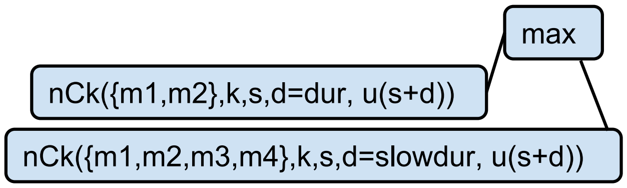
Unconstrained: Unconstrained is the most primitive type of job that has no placement preference and derives the same amount of benefit from an allocation of any k servers. It can be represented with a single "n Choose k" primitive, choosing k servers from the whole cluster serving as the equivalence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

GPU: GPU preference is one of the simplest forms of non-combinatorial constraints. In addition to the SLA parameters above, users also 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 user-specified slowdown factor[5]. This translates into a tetrisched max expression with two branches corresponding to k GPU-enabled servers and k servers anywhere (see Fig. 6(a)). The max expression carries the semantics of an "OR" operator. This pattern is repeated for each possible start time. 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.

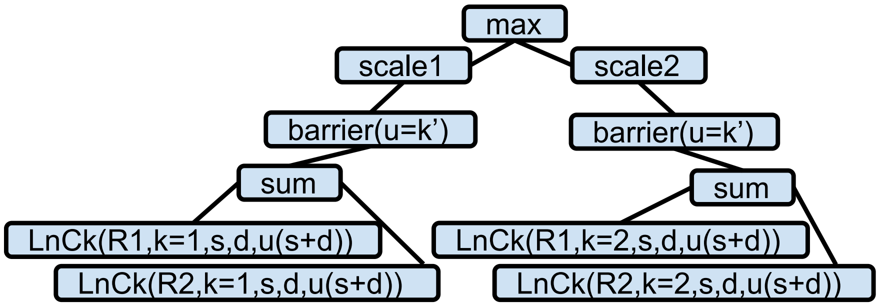
MPI: Rack locality is a prime example of a combinatorial constraint. For example, many MPI workloads are known to run faster with locality. Here, users request k servers on the same rack and get slowed down if their allocated servers are on different racks. This translates into a tetrisched max expression with a branch per rack plus a branch for running anywhere. 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 spreading servers across failure domains benefit from this type of scheduling. The tetrisched expression for this type of job (Fig. 6(b)) 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 availability, users specify utility degradation as a function of availability. This function

[5]We envision estimated job runtimes and slowdown factors being provided by profiling tools rather than actual user knowledge.


max
nCk({m1,m2},k,s,d=dur, u(s+d))
nCk({m1,m2,m3,m4},k,s,d=slowdur, u(s+d))

(a) GPU



(b) Availability

Figure 6: Algebraic utility expressions for GPU and Availability jobs.

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 across all racks (used as an example of a failure domain). 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. The translation from k' to job's budget is performed in the scale operator along with utility attenuation due to respective availability degradation. HA jobs are noteworthy as they combine two objectives: queueing delay and availability. In Section 7.4, we evaluate the extent of user control over their relative importance by tuning the user-defined utility function (Fig. 4).

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. Similar 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 tetrisched 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-utility pair (p,q), where p is the number of HDFS storage nodes and q is the number of non-HDFS nodes.

6 Experimental Setup

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

6.1 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

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 1: Workload compositions used in results section.

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.

6.2 Workload Configuration

The experiments use four workload types described in Sec. 5.3: GPU, HDFS, HA (high availability), and Unconstrained. In this section, we describe pertinent parameters that affect the workload.

Definition of ρ: An important parameter that affects the impact of scheduling is load (ρ). In all our 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 an unoptimal placement may increase the effective load on the system, if the job is slowed down. We define **slowdown** in Table 2.

Workload Composition: Workloads are often composed of a heterogeneous mixture of job types. We experimented with many different proportions of workload types, as well as a broad range of settings of the other parameters. Table 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 spacial 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 [10]. 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 unoptimal resources plus the desired completion time. Thus, it is possible to extract positive value from jobs even if they are running on unoptimal 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,

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 2: Metrics and parameters used in results section.

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 we show in Sec. 7.4 how these two parameters can be tuned by the user to prefer increased availability or reduced queueing delay.

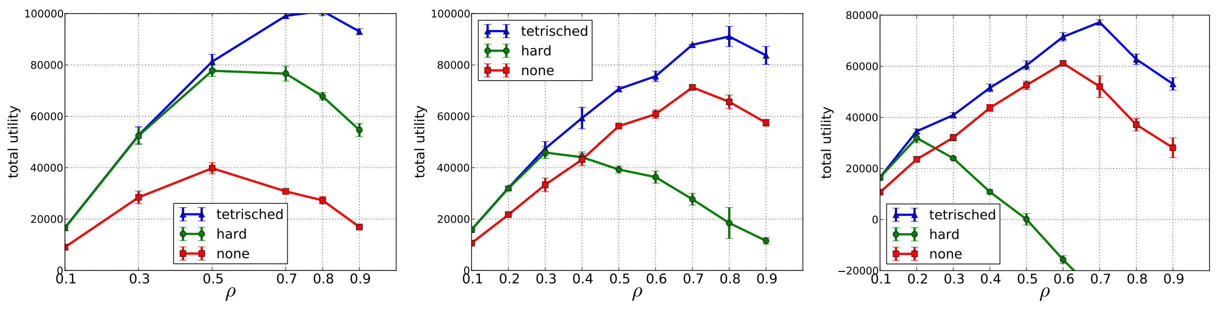
Traces: We generate traces based on load, workload composition, job type, and burstiness. The trace file for each contributing workload type is generated independently. The traces include arrival time, estimated job duration, and the number of servers requested. Arrival times are generated based on each 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 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.

6.3 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 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 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. 10(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.



(a) W2, slowdown=3 (b) W3, slowdown=1.5 (c) W4, slowdown=1.5

Figure 7: `tetrisched` outperforms `hard` and `none` as cluster load(ρ) increases. Graphs 7(a), 7(b), and 7(c) correspond to workload compositions in Table 1 with a Poisson inter-arrival process ($C_A^2 = 1$) and schedulers using 15-minute plan-ahead.

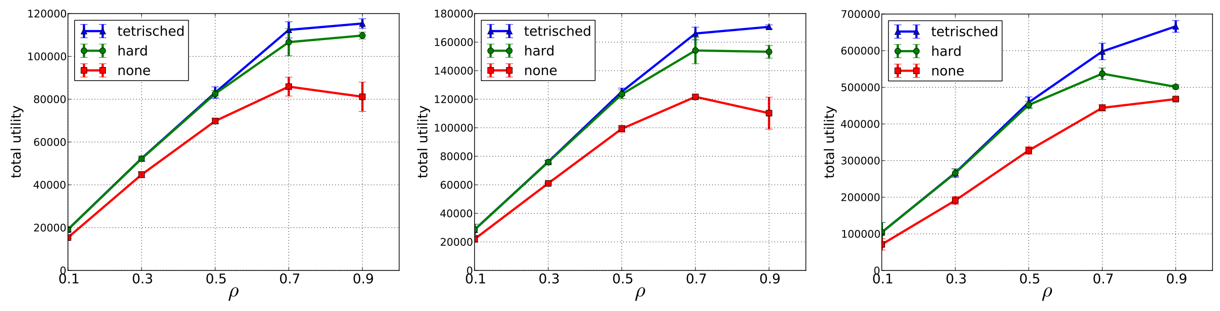
7 Experimental Results

This section evaluates `tetrisched` across a wide range of workload characteristics and mixes. Sec. 7.1 examines overall performance, showing that `tetrisched` consistently improves utility over `hard` and `none`, by as much as a factor of 2 at high load, where scheduling has the most impact. Sec. 7.2 examines the performance metrics behind utility, showing that `tetrisched`'s superior utility is often a consequence of sometimes accepting less-preferred resources in order to avoid dropping as many jobs as other policies. Sec. 7.3 examines the importance of plan-ahead, which is a key differentiator between `tetrisched`, which supports plan-ahead, and `alsched`, which does not. We find that plan-ahead can improve the performance of `tetrisched` by a factor of 3 or more over `alsched`, particularly when the slowdown factor is high and job arrivals are bursty. Sec. 7.4 illustrates how the user can control the tradeoff between higher availability and lower queueing delay using the interface provided by the wizard. Sec. 7.5 shows that `tetrisched` is surprisingly robust to user error in specifying job durations.

All graphs plot median data points from a set of 6 runs per data point. Error bars are one ± median absolute deviation (MAD). Observed variability comes primarily from regenerating trace files for each of the 6 sets of parameter sweeps. Per each workload composition (Table 1), the number of runs in each parameter sweep depended on the number of scheduling policies compared (3), slowdown factors (5), plan-ahead windows (4), load factors (6-9), and levels of burstiness (4). Thus, a typical single parameter sweep required anywhere from 1440 to 2160 simulation runs, averaging more than ten thousand simulation runs per workload composition.

7.1 Overall tetrisched performance

Benefits of spacial flexibility: Figures 7 and 8 compare utility as a function of load (ρ) for the 4 workload compositions in Table 1. For each, `tetrisched` is superior to `hard` and `none`, while the exact relationship between `hard` and `none` depends on the mix. For W2 (Fig. 7(a)), we see that `hard` performs similarly to `tetrisched` when load is low, as there's enough preferred resources to give most jobs their first choices. As load increases beyond $\rho = 0.5$, however, `hard` starts to deteriorate as expected since all preferred resources saturate at $\rho \approx 0.5$. HA jobs in the mix prefer thin placement across most racks, cutting into the GPU and HDFS capacity. `hard` policy takes this as a requirement, increasing contention for preferred resources. Specifically, 50% HA jobs at $\rho = 0.5$ contend for 25% of the cluster, cutting into 25% of GPU rack capacity and leaving only $\frac{3}{4} * \frac{1}{4} = \frac{3}{16}$ of the total cluster capacity to GPU jobs. Since the `hard` policy insists on preferred placement, the GPU workload starts to saturate GPU racks at $\rho = 0.5$, squeezed by HA jobs. The



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

Figure 8: `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.

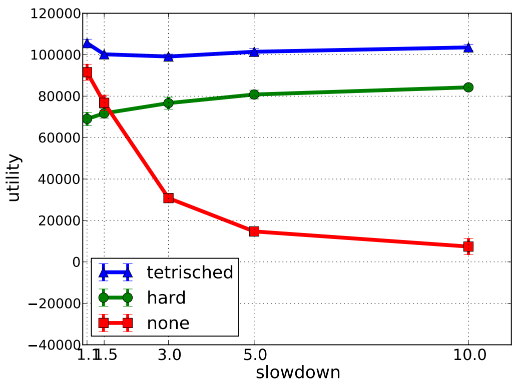
situation is similar for HDFS. Tetrisched, on the other hand, extracts benefit from the spacial flexibility of HA jobs as well as GPU jobs. Scheduling HA jobs with up to 2 tasks per rack (see Sec. 5.3) eases that contention, allowing all three job types to benefit from non-preferred spacial capacity. As a result, we observe the inflection point for `tetrisched` occurring at a higher load of $\rho = 0.7$.

Fig. 7(b) shows results for a less spatially balanced W3 mix. The more unbalanced the composition is, the further left the inflection point moves for `hard` (the earlier `hard` starts failing). At $\rho = 0.4$, the workload is already close to the 0.25 cluster GPU capacity, given that GPU is 50% of the workload in this composition. As we move from Fig. 7(a) to 7(b) to 7(c) (with GPU fraction increasing from 25% to 50% to 100%), the workload becomes increasingly unbalanced, shifting the inflection point for `hard` further left, from $\rho = 0.5$ to $\rho = 0.4$ to $\rho = 0.2$.

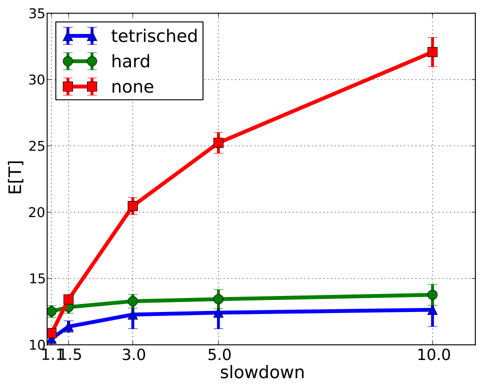
`none` performs quite well at sufficiently small slowdowns, since there is little penalty caused by ignoring preferences. Unaware of constraints, it enjoys the same access to spare capacity as `tetrisched` does. For slowdown factor of 1 and workload compositions consisting entirely of completion oriented jobs, we expect and observe `none` behave the same as `tetrisched`. As shown in Fig. 7(a), it quickly loses this benefit with an increase in load. In all cases, however, `tetrisched` extracts benefit from the spare capacity and outperforms `hard` and `none` by up to a factor of 2.

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. 8). 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. 8 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 "picky" jobs. Recall that all completion oriented jobs are described by the *uduf*s in Fig. 4. Budget is the maximum utility a job can extract from the cluster. Relative budget differences, therefore, translate into relative job importance, as the scheduler will favor jobs that yield higher utility. In Fig. 8(a), all jobs have the same budget. We increase the budget for "picky" jobs in Fig. 8(b) and 8(c) . As a result, `tetrisched` achieves higher relative gains if 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 flexible scheduling.

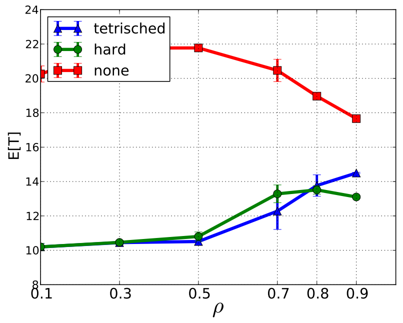


(a) utility

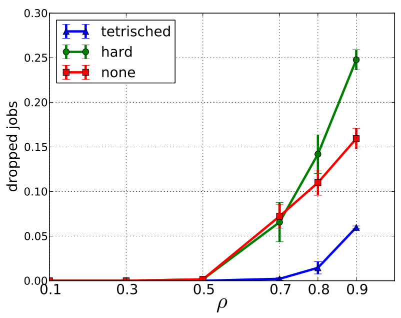


(b) mean response time

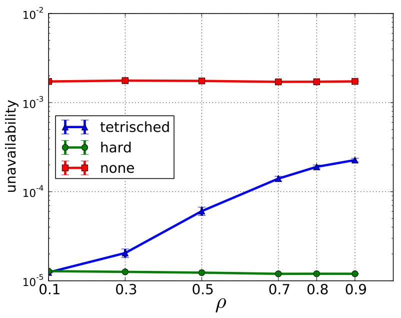
Figure 9: Utility and response time as function of slowdown. Workload is same as Fig. 7(a) at load ($\rho = 0.7$).



(a) mean response time



(b) dropped jobs



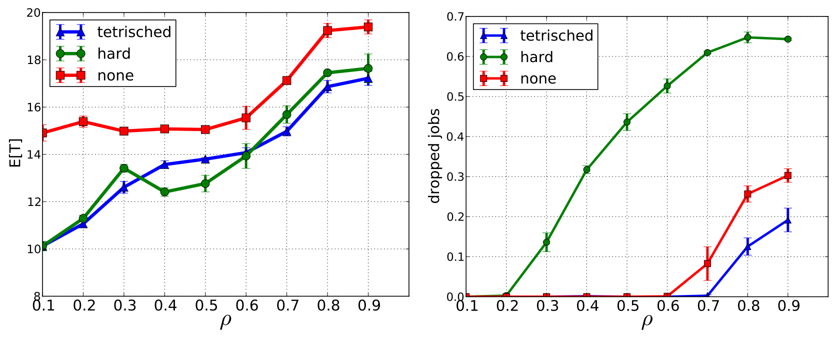
(c) unavailability

Figure 10: 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. Workload is identical to Fig. 7(a).

Effect of slowdown: Slowdown (see Table 2) is an important factor that affects the relative performance comparison of the three policies. For slowdowns as low as 1.1, `none` performs well (see Fig. 9(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. 9(a)). Lastly, slowdown affects `tetrisched` mean response time, $E[T]$, as well. Note that in Fig. 9(b) `tetrisched` actually surpasses `hard` with respect to $E[T]$, particularly at slowdown factors of 3 or less.

7.2 Under the hood

What makes `tetrisched` yield 2x better utility in Fig. 7(a)? Fig. 10 shows the underlying metrics that affect the difference in utility. In Fig. 10(a), we see, surprisingly, that the **average response time**, $E[T]$, (defined in Table 2) of `hard` matches that of `tetrisched`. In Fig. 10(b), we see that the utility difference is due to `hard` dropping significantly more jobs. Note that `hard` will *always* cause jobs to wait for preferred resources, if not immediately available. When resources become available, `hard` is likely to pick younger jobs to place



(a) mean response time (b) dropped jobs

Figure 11: Under the hood of Fig. 7(c) (Workload composition W4). `tetrisched` drops fewer jobs while providing response times akin to `hard`. Lower is better.

on preferred resources, as they have the highest value (see Fig. 4). As older jobs eventually reach their response time deadlines, they are dropped. `hard` thus achieves good response times at the expense of dropped jobs. `tetrisched` matches the response time performance of `hard`, but manages to drop a lot fewer jobs by exploiting the availability tradeoff for HA jobs (see Fig. 10(c)). Note that this tradeoff only occurs at high load when the cluster is contended.

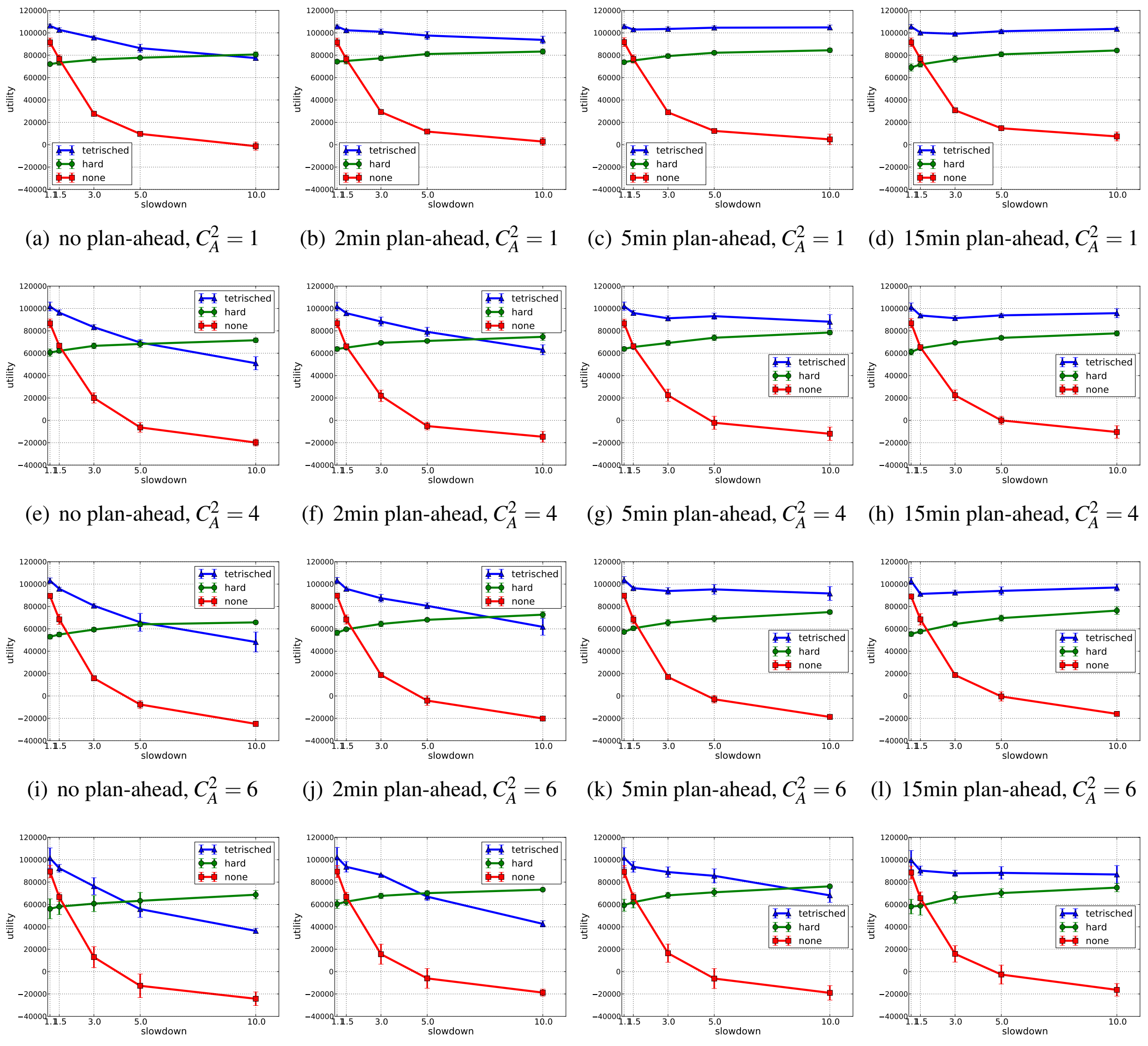
`tetrisched` extracts benefit from spare capacity. This is especially pronounced in homogeneous compositions, when tradeoffs cannot be made between multiple job types. In Fig. 11(a), we see that `tetrisched` continues to match the response time of `hard`, but it drops far fewer jobs in the process as shown in Fig. 11(b). In this case, `tetrisched` is able to drop fewer jobs as it can use the less optimal resources, whereas `hard` stubbornly insists on using preferred resources only.

7.3 Benefits of time-aware placement

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. 12 plots utility for `tetrisched`, `hard`, and `none` as slowdown increases from 1.1 to 10. `tetrisched` without plan-ahead (Fig. 12(a),12(e),12(i),12(m)), positioned in the first column of Fig. 12, 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 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. 12(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. 12(a),12(e),12(i),12(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. 12. Plan-ahead helps tetrisched handle this increasing temporal imbalance, however, illustrated with a gradually improving relative performance, as we horizontally scan any



(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, $C_A^2 = 6$

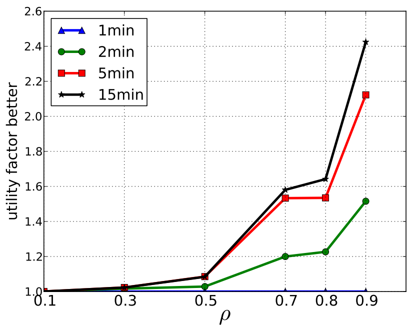
(m) no plan-ahead, $C_A^2 = 8$ (n) 2min plan-ahead, $C_A^2 = 8$ (o) 5min plan-ahead, $C_A^2 = 8$ (p) 15min plan-ahead, $C_A^2 = 8$

Figure 12: 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`).

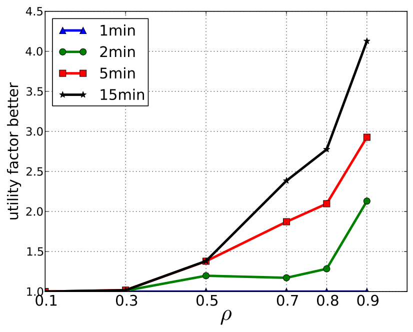
of the 4 subfigure rows. Fig. 13 highlights the factors of improvement tetrisched achieves from plan-ahead. Specifically, the highest factor difference for `tetrisched` between Fig. 12(p) and Fig. 12(m) is plotted in Fig. 13(b) as 2.4x ($\rho = 0.7$), and we see even higher factors of improvement for higher loads.

7.4 Exerting control through utility

While utility is, by design, the single primary objective for which `tetrisched` optimizes, user-specified utility functions can serve as expressive tools to guide the scheduler in the tradeoff space towards desired

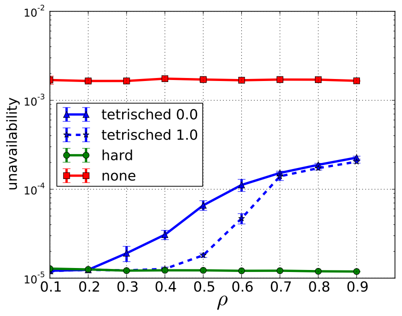


(a) slowdown=5

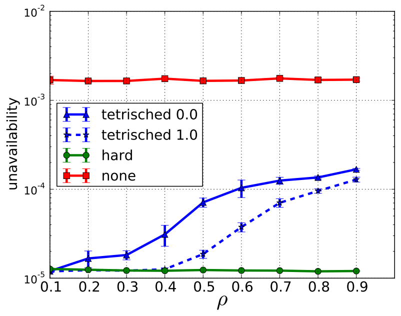


(b) slowdown=10

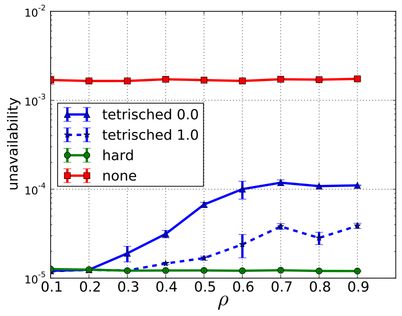
Figure 13: This graph shows factors of improvement for the `tetrisched` policy over `alsched` as a function of cluster load (ρ) and plan-ahead windows.



(a) 1x availability job budget



(b) 2x availability job budget



(c) 10x availability job budget

Figure 14: User-defined utility functions provide a mechanism to control the tradeoff between availability and delay sensitivity. This experiment is run with the W2 mix. Higher HA job budgets give them preference over the other GPU and HDFS jobs, particularly at high loads (ρ).

outcomes. Specifically, a user-defined utility function (uduf) maps a desired completion time or queueing delay to utility. For HA jobs, that utility value is additionally scaled by a factor $\in [0;1]$ computed as a function of expected failure probability for a given placement. Fig. 14 shows that, in cases of multiple such considerations, uduf's provide knobs to tune user preference for one such consideration (e.g., queueing delay) over the other (e.g., failure probability).

Turning the uduf knob away from queueing delay as the primary consideration ("tetrisched 0.0") to higher availability ("tetrisched 1.0") increases the separation between blue triangle curves in Fig. 14 – a delta most visible at higher ρ with an increased budget (Fig. 14(c)). Whereas in Fig. 14(a), under high load, the scheduler traded off availability for the utility it extracted from scheduling competing workload, we see in Fig. 14(c) higher availability is achieved. Furthermore, availability increases both across all ρ and specifically for the "tetrisched 1.0" case, which sets the HA-sensitivity to the maximum for availability jobs. The takeaway is that user-defined utility functions, as exemplified by availability jobs, offer several control knobs, such as budget and desired queueing delay deadline, empowering users to control the tradeoffs made by the scheduler.

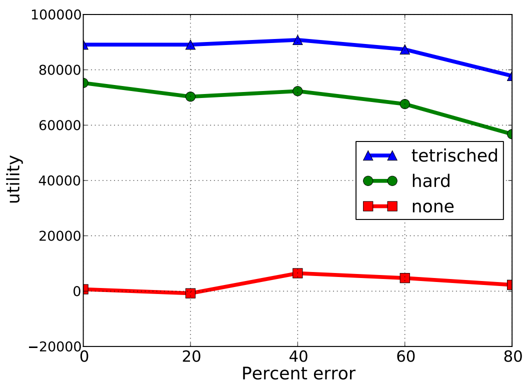


Figure 15: Effect on utility as users are more erroneous about duration estimates. `tetrisched` is robust to error in duration estimates. The average error is kept constant so that load ($\rho = 0.7$) remains constant. Percent error is calculated as the root mean square error divided by the average duration. This experiment uses the W2 mix with some burstiness ($C_A^2 = 4$).

7.5 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 15 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.

8 Previous Work

Recent studies of datacenter traces [24, 21] have highlighted the facts that (a) datacenter infrastructures are often heterogeneous, (b) workloads are heterogeneous, and (c) scheduling constraints often accompany heterogeneous workloads. Tetrisched builds upon a tremendous amount of prior scheduling research and extends it by addressing the scheduling of diverse workloads on heterogeneous servers while comprehending soft constraints expressed in both space and time as well as combinatorial and gang scheduling constraints.

Datacenter Scheduling: Recent work on datacenter schedulers has studied multi-level scheduler designs [23], dynamic resource partitioning in heterogeneous datacenters [13], fairness [11], and placement considerations [12, 20, 14]. Omega's [23] key contribution is enabling multi-scheduler designs for datacenter scheduling. Tetrisched is complementary to this objective. Omega explicitly supports hard scheduling constraints, as evidenced by the authors' high fidelity simulator support for them [23]. But, there's no mention of preferential placement constraints in that work. Mesos [13] addresses the problem of dynamically partitioning resources in heterogeneous datacenters via a multi-level scheduler design. Mesos defers the complexity of constraints to framework-specific schedulers by design. As such, any gang scheduling or preferential constraints can only be accomplished via hoarding.

Other cluster scheduling work has painted fairness as the most important metric [12, 11, 8]. While Choosy [12] does consider constraints, it accommodates neither soft constraints nor combinatorial constraints. While fairness is important in some situations, such as scheduling on some academic clusters, results-oriented measurements (e.g., throughput, availability, or utilization) dominate in other environments. Consequently, our work focuses on exploring the use of utility functions as a means of optimizing over potentially diverse results-oriented user objectives.

Condor ClassAds [20] supports behavior akin to what we call "soft constraints". The ClassAd mechanism allows resource providers to declaratively describe each individual resource with expressions built on a collection of resource-descriptive attributes (e.g., disk, memory, MIPS, arch, name). Resource consumers, through the same mechanism, express their resource allocation preferences by submitting a classad describing their respective attributes as well as, optionally, specifying *Constraint* and *Rank* expressions for desired resources. The former serves as a hard constraint—a filter on allowable resources. The *Rank* can be thought of as the expression of preference over individual machines. The matchmaking algorithm then evaluates each consumer classad against each resource classad and picks the highest ranked resource. This approach has two shortcomings relative to tetrisched. First, it relies on an a priori agreed upon static set of attributes—the underlying expression vocabulary. Tetrisched instead "compiles" a heterogeneous set of objectives to a common algebraic expression scheduler language that *need not change* as new resource attributes, job characteristics, or user objectives appear. Second, it is fundamentally bilateral, matching a single job to a single machine. That leads to lack of support for combinatorial constraints and gang scheduling, which tetrisched enables.

A family of work including Quincy [14], MapReduce [9], and ABACUS [7] provides support for specific types of placement preference with a hard-coded preference structure. Quincy [14], in particular, offered an interesting range of support for data locality, fairness, and starvation-freedom. It represents the scheduling problem as a network flow problem, capitalizing on such problems' good worst-case time complexity properties. Quincy achieves admirable cost-savings by addressing one specific type of placement preference – data locality. Tetrisched generalizes that to arbitrary placement preferences, by providing support for associating preferences with arbitrary subsets of resources. The latter is key to supporting combinatorial constraints (rack-affinity, failure domain anti-affinity) as well as gang scheduling in a single scheduling cycle. This helps tetrisched avoid hoarding (and associated potential for deadlock) needed by other systems, including Mesos [13], to achieve the same. Additionally, none of the systems mentioned above have support for plan-ahead. Tetrisched extracts factors of improvement (Fig. 13), in some cases, from its ability to consider future placements and does so in a manner resilient to up to 3.4x runtime mis-estimation (Section 7.5).

Apache Hadoop YARN [27] is an open source cluster resource management system with multi-framework support. It enables resource allocation support for a heterogeneous mix of programming frameworks. YARN's ResourceManager (RM) exports a resource request interface [3], enabling an ApplicationMaster (AM) to submit resource requests. Similarly to [14],[9], and [7] above, however, YARN focuses its soft constraint support on a specific type of preference—data locality, with preference fallback structure embedded in the ResourceRequest API [3]. The latter makes it possible to specify specific servers, racks, and wildcard placement preferences. Further, while [27] claimed support for gang-scheduling, we found no details of that in the paper. In fact, YARN API examples [5] and API documentation [3, 2, 4] revealed no way of specifying that requested containers must be gang-allocated, explicitly stating that "all containers may not be allocated at one go" [5]. Granted, as is the case with several other systems we've looked at, gang-allocation is possible through hoarding allocated containers at the AM level. Thus, YARN offers support for data locality with a narrow, predetermined preferential structure, no support for combinatorial constraints, and no plan-ahead.

alsched: The closest prior work, by far, to tetrisched is our vision paper describing a system called `alsched` [26]. This paper sketched the idea of using utility functions to schedule jobs with soft constraints (using a more primitive language for utility functions and a crude bin-packing placement algorithm rather than tetrisched's complete MILP formulation). However, alsched ignored the time dimension: its utility functions did not include time considerations, and it did not (and could not) use plan-ahead in scheduling. (Recall, from Figures 12(a), 12(e), 12(i) and 12(m), alsched's inability to wait for preferred resources to become available). Further, while alsched hypothesized the possibility of a wizard for user objectives, tetrisched introduces the first design and realization of such a wizard. Finally, the present paper provides a thorough evaluation of utility function-based scheduling, which was lacking in the earlier work.

Utility functions: Wilkes [28] provides a tutorial and partial coverage of utility theory in scheduling, particularly for managing tradeoffs between services with distinct service level objectives. Kelly [15] describes utility-directed allocation using combinatorial auctions and solving the scheduling problem as a multi-dimensional multi-choice knapsack problem. Tetrisched employs a richer formulation that supports specifications over arbitrary subsets of resources (rather than only quantities of each resource type) as well as those enhanced by time specifications; further, tetrisched supports plan-ahead.

Most proposed uses of utility functions for scheduling [17, 15, 25, 18, 16] have focused on quantifying the value of resource quantities to each consumer, informing scheduling decisions when tradeoffs must be made between them. Other papers use utility functions to model the cost of resource allocation or otherwise help the scheduler arrive at allocations that optimize for the overall utility of the cluster [25, 18, 16]. Recently, Jockey [10] used utility functions to map the duration of job execution to a utility value that decreases and potentially drops below zero as the duration exceeds predetermined deadlines– a concept tetrisched borrows for user-defined utility functions. However, Jockey makes only local optimization decisions per job and does not reason about tradeoffs between multiple latency SLO jobs.

9 Conclusion

Tetrisched effectively schedules heterogeneous resources among a collection of diverse applications. Given job-specific utility functions quantifying tradeoffs among preferences, plus estimated job runtimes, it plans resource assignments to maximize overall utility. Simulation results of a 1000-node cluster show that it consistently outperforms competing schemes that fail to consider either preferences or estimated runtimes, across a wide range of workload scenarios and even with estimate inaccuracy. As a result, tetrisched is a promising scheduling solution for heterogeneous datacenters and cloud infrastructures.

References

[1] Hadoop, 2012. http://hadoop.apache.org.

[2] Apache Hadoop main 2.2.0 API: Class Resource, Nov 2013. http://hadoop.apache.org/docs/current/api/org/apache/hadoop/yarn/api/records/Resource.html.

[3] Apache Hadoop main 2.2.0 API: Class ResourceRequest, Nov 2013. http://hadoop.apache.org/docs/current/api/org/apache/hadoop/yarn/api/records/ResourceRequest.html.

[4] Apache Hadoop Main 2.2.0 API: Interface ApplicationMasterProtocol, Nov 2013. http://hadoop.apache.org/docs/current/api/org/apache/hadoop/yarn/api/ApplicationMasterProtocol.html.

[5] Hadoop MapReduce next generation - writing YARN applications, Nov 2013. http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/WritingYarnApplications.html.

[6] IBM CPLEX Optimizer, 2013. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[7] 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.

[8] 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.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[15] Terence Kelly. Utility-directed allocation. Technical Report HPL-2003-115, Internet Systems and Storage Laboratory, HP Labs, June 2003.

[16] 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.

[17] 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.

[18] 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.

[19] Cade Metz. Why even google will embrace cellphone chips in the data center, 2013. http://www.wired.com/wiredenterprise/2013/05/google-jason-mars/, Wired.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] John Wilkes. Utility functions, prices, and negotiation. Technical Report HPL-2008-81, HP Labs, July 2008.

A Appendix

A.1 Partitioning algorithm

As defined in Sec. 3.1, *equivalence classes* contain machines that are equivalent in the context of a single tetrisched leaf node expression, and *partitions* contain machines that are equivalent in the context of all jobs and expressions. For example, Fig. 16 illustrates how three overlapping equivalence classes (circles) are related to the partitions. Algorithm 1 converts a set of equivalence classes into a set of partitions. The key insight is that a partition is uniquely identified by the set of equivalence classes of which it is a member. Each bitvector position corresponds to a given equivalence class. To illustrate this, Fig. 16 labels each partition with a bitvector encoding the equivalence classes it is a member of. Our algorithm creates these bitvectors and assigns partition numbers to each unique bitvector.

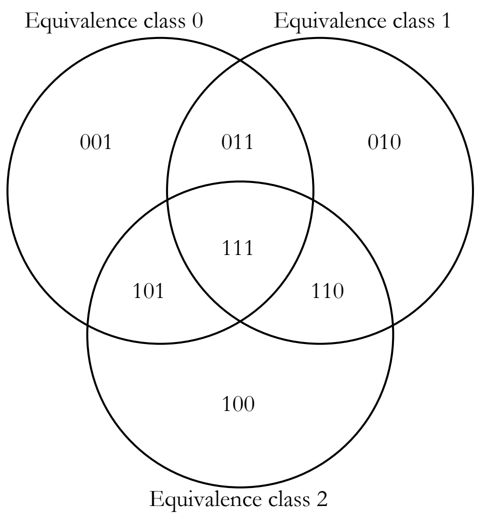


Figure 16: The circles represent equivalence classes that have overlapping machines. Each region is a partition and is uniquely determined by the bitvector of equivalence classes.

A.2 MILP generation algorithm

In Sec. 4, we present the intuition behind representing our scheduling problem as an MILP problem. In this section, we present the algorithm for translating a tetrisched expression into an MILP problem. As tetrisched expressions are built as expression trees, we perform the translation recursively on each branch of the expression tree. Each branch roughly corresponds to a choice of whether a set of resources is consumed or not, and we associate an indicator variable to represent if the branch is chosen. In our generation function, the indicator variable is passed as one of the parameters. We do this instead of creating an indicator variable for each expression because it allows the min operator to pass the same indicator variable to each child expression. This ensures all its branches are either chosen or not chosen without needing extra constraints.

The min operator is also different in that it creates a variable corresponding to the minimum utility across its child branches. This variable is constrained to be less than the utility from each branch and returned as the objective function to maximize, thus making the variable equal to the minimum utility. Unlike the min operator, the max operator does not need to create an additional variable, because it can simply limit the child indicator variables to choose at most one branch. By returning the sum of child objective functions, the solver will attempt to choose the one branch that maximizes utility, subject to capacity constraints.

The last set of variables we use are partition variables, which correspond to the number of machines that a particular nCk or LnCk leaf node consumes within a specific partition. Partition variables are used in demand and supply constraints. Demand constraints ensure leaf nodes get their requested number of machines across their allowable partitions. Supply constraints ensure the number of allocated machines is limited by the cluster capacity. As supply constraints depend on the entire expression tree, we only collect partition usage info in the gen function in $cap(x,t)$, which stores the set of partition variables that are used for a given partition x at a given time t. After the MILP generation is performed, we add the supply constraints so that each set of partition variables sums to less than the capacity of the partition x at time t (e.g., number of machines in partition x minus the number of machines in partition x that are expected to be in use at time t).

```
partition: equivClasses → partitions
func partition(equivClasses):
    // Create bitvectors for each machine corresponding to the equivalence classes
    // of which the machine is a member
    bitvectors := new array of bitvector
    foreach index, equivClass in equivClasses :
        foreach machineId in equivClass :
            bitvectors[machineId].setBit(index)
    // Machines with the same bitvector get assigned the same partition number
    partitions := new array of int
    hashtable := new hashtable from bitvector to int
    nextPartition := 0
    for machineId := 0 to cluster size :
        partition, found := hashtable.find(bitvectors[machineId])
        if ! found :
            // Bitvector hasn't been seen before; assign new partition number to bitvector
            partition = nextPartition
            hashtable.insert(bitvectors[machineId], nextPartition)
            nextPartition++
        partitions[machineId] = partition
    return partitions
```

Algorithm 1: Partitioning algorithm

```
gen: (tetrisched expr tree, indicator var) → objective function
func gen(expr, I):
    switch expr :
        case $nCk(partitions, k, u, s, dur)$
            foreach $x$ in partitions :
                $P_x$ := integer variable // Create a partition variable per partition in equiv class
                for $t := s$ to $s + dur$ :
                    Add $P_x$ to $cap(x,t)$ // Add partition variable to cap to track supply constraints
            Add constraint $\sum_x P_x = k * I$ // (Demand) Ensure this node gets k nodes if chosen
            return $u * I$ // Return utility if chosen
        case $LnCk(partitions, k, u, s, dur)$
            foreach $x$ in partitions :
                $P_x$ := integer variable // Create a partition variable per partition in equiv class
                for $t := s$ to $s + dur$ :
                    Add $P_x$ to $cap(x,t)$ // Add partition variable to cap to track supply constraints
            Add constraint $\sum_x P_x \leq k * I$ // (Demand) Ensure this node gets up to k nodes
            return $u * \sum_x \frac{P_x}{k}$ // Return scaled utility based on the number of allocated nodes
        case $sum(t_1, ..., t_n)$
            for $i := 1$ to $n$ :
                $I_i$ := binary variable // Create indicator variable for each branch
                $f_i$ = gen($t_i$, $I_i$)
            Add constraint $\sum_i I_i \leq n * I$ // Ensure that no branches are chosen if $I = 0$
            return $\sum_i f_i$
        case $max(t_1, ..., t_n)$
            for $i := 1$ to $n$ :
                $I_i$ := binary variable // Create indicator variable for each branch
                $f_i$ = gen($t_i$, $I_i$)
            Add constraint $\sum_i I_i \leq I$ // Ensure that at most one branch is chosen
            return $\sum_i f_i$
        case $min(t_1, ..., t_n)$
            $U$ := continuous variable // Create variable representing min utility
            for $i := 1$ to $n$ :
                $f_i$ = gen($t_i$, $I$) // Choose branches based on the same indicator variable $I$
                Add constraint $U \leq f_i$ // Ensure $U$ is less than the min utility
            return $U$ // Given the constraints, maximizing $U$ makes $U$ equal to the min utility
        case $scale(t, s)$
            return $s$ * gen($t$, $I$) // Scale the objective function by $s$
        case $barrier(t, bar)$
            $f$ = gen($t$, $I$)
            Add constraint $bar * I \leq f$ // Ensure the child expression meets barrier constraint
            return $bar * I$ // Barrier also caps the utility at bar
$I$ := binary variable // Dummy indicator variable
$f$ = gen(expr, $I$)
foreach $x$ in partitions :
    for $t := now$ to $now + horizon$ :
        // Add supply constraints for each partition $x$ and time $t$
        Add constraint $\sum_{P \in cap(x,t)} P \leq$ size($x$) - in_use($x$, $t$)
solve($f$, constraints)
```

Algorithm 2: MILP generation algorithm