alsched: Algebraic Scheduling of Mixed Workloads in Heterogeneous Clouds

Alexey Tumanov
Carnegie Mellon University
atumanov@cmu.edu

James Cipar
Carnegie Mellon University
jcipar@cmu.edu

Michael A. Kozuch
Intel Labs
michael.a.kozuch@intel.com

Gregory R. Ganger
Carnegie Mellon University
ganger@ece.cmu.edu

ABSTRACT

As cloud resources and applications grow more heterogeneous, allocating the right resources to different tenants' activities increasingly depends upon understanding tradeoffs regarding their individual behaviors. One may require a specific amount of RAM, another may benefit from a GPU, and a third may benefit from executing on the same rack as a fourth. This paper promotes the need for and an approach for accommodating diverse tenant needs, based on having resource requests indicate any soft (i.e., when certain resource types would be better, but are not mandatory) and hard constraints in the form of composable utility functions. A scheduler that accepts such requests can then maximize overall utility, perhaps weighted by priorities, taking into account application specifics. Experiments with a prototype scheduler, called **alsched**, demonstrate that support for soft constraints is important for efficiency in multi-purpose clouds and that composable utility functions can provide it.

Categories and Subject Descriptors

D.4.7 [**Operating systems**]: Organization and design—*Distributed systems*

General Terms

Design

Keywords

cluster scheduling, cloud computing

1. INTRODUCTION

Some applications benefit from specific characteristics in the machine resources on which they execute [12, 13]. For example, a sort program may need a certain amount of RAM and a certain CPU speed to achieve acceptable performance. A backup replica of a service may need to be run on a different rack than the primary in order to maximize availability. A specialized image processing code may only be able to run on a machine with a GPU. A data analysis task may achieve higher performance if executed on the server that stores the data in question on a local disk. Many other examples exist.

When clusters are dedicated to particular application types, schedulers can accommodate such benefits by either having hard-coded understanding (e.g., locality awareness in Hadoop [1]) or ignoring such benefits (if all machines satisfy them equally) when assigning tasks. With cloud computing, however, such approaches will not work. Diverse mixes of applications will share cloud infrastructures, and their varied specific needs must be accommodated if promised efficiency gains are to be realized. Worse, those needs will not be known *a priori*—they must be communicated with resource requests made to the cluster scheduler, which must then be able to make good decisions accordingly.

One approach is for resource consumers to specify zero or more *hard constraints* with each request, based on some predetermined attribute schema understood by the cluster scheduler [12, 13]. Such constraints could serve as a filter on the set of machines, enabling identification of the subset that are suitable for the corresponding request. But, this approach ignores an important issue: in many cases, the 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. We refer to such cases as *soft constraints*. Treating them as hard constraints can lead to unnecessary resource under-utilization, and ignoring them can lead to lower efficiency, making them an important consideration for cloud schedulers.

This paper proposes a specific approach for accommodating soft constraints, as well as hard constraints and general machine heterogeneity. In our model, each job submitted for processing is accompanied by a resource request, which is expressed as utility functions in the form of algebraic expressions indicating what benefit would be realized if particular resources were assigned to it. The expressions can indicate specific machines, but can also use "n Choose k" primitives over resource sets. The resource sets can be indicated via attributes (as described above for hard constraints) or explicit listings of specific machines. We believe that such utility functions are flexible and expressive enough to capture hard and soft constraints of interest, while also being sufficiently simple to construct that low-level software (e.g., libraries) can translate an application's needs into them effectively. Given a set of resource constraints specified as such expressions, a scheduler can search for an optimal placement to maximize utility.

We have implemented such a scheduler, called *alsched*, and a

SOCC'12, October 14-17, 2012, San Jose, CA USA

simulator for evaluating its effectiveness. We describe a number of hypothetical applications with soft constraints corresponding to real application characteristics, and use simulated mixes of them to evaluate both the need for and alsched's support of soft constraints. The results show that soft constraints should be explicitly supported; treating them as either hard constraints or no constraints, instead of soft constraints, often leads to wasted resources and higher task completion delays. The results also show that alsched is effective at providing that support, quickly finding good schedules that exploit the soft constraints to improve utility.

The remainder of this paper further evangelizes the importance of soft constraints in cloud scheduling and our approach to providing for them. Section 2 discusses related work and describes a range of soft constraints that are evident in existing environments. Section 3 details the interface that consumers use to specify resource requests, including soft and hard constraints, and gives examples of its use. Section 4 describes our prototype alsched implementation. Section 5 evaluates the need for soft constraints and alsched's effectiveness in handling them.

2. BACKGROUND AND MOTIVATION

Cluster scheduling has been studied and practiced so extensively that we will not attempt to summarize it all. The vast majority of it, however, targets much less diverse environments than multi-purpose clouds. Analysis of recently released Google trace data [12] indicates that early clouds already exhibit heterogeneity of hardware resources and especially of job characteristics, including resource requirements and constraints. Researchers are promoting even more use of specialized platform mixes [12]. New scheduling approaches will be needed to handle such heterogeneity, and here we focus on the central issue of supporting job-specific trade-offs (i.e. soft constraints) regarding which resources are assigned to executing a given job. This section discusses prior work related to our approach as well as the extensive evidence that soft constraints will play an important role in cloud schedulers.

2.1 Related work

Researchers have proposed use of utility functions in cluster scheduling in various ways and explored potential benefits. Wilkes [15] provides a useful tutorial and partial coverage of utility theory in scheduling, particularly for managing tradeoffs between services with distinct service level objectives (SLOs). Utility-based approaches, rooted in sound economic theory, are an attractive way to manage complex tradeoffs in resource allocation among consumers.

Generally speaking, utility functions would be associated with each consumer's resource request, and the scheduler would allocate resources to requests so as to maximize overall utility. For example, Kelly [5] describes utility-directed allocation using combinatorial auctions and solving the scheduling problem as a multi-dimensional multi-choice knapsack problem. At a high level, our approach is similar, but focuses on using utility functions to quantify the individual values of the wide range of soft constraints.

Most proposed uses of utility functions for scheduling have focused on quantifying the value of resource quantities to each consumer, informing scheduling decisions when tradeoffs must be made between them—in particular, deciding which consumer doesn't get what it requested, when there are not enough resources to satisfy them all. There are many such examples, ranging over the last two decades [9, 5, 14, 10, 6]. As a recent example, Jockey [3] uses 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.

Though promising, the use of utility functions has never quite taken hold [8]. We believe that cluster schedulers have succeeded without them, primarily by being able to avoid too much diversity in tradeoffs (allowing hard-coding of sufficient support for them) and too much resource contention (often allowing all high-priority consumers to obtain desired resources). For example, Hadoop's JobTracker tries to schedule map tasks on machines with a copy of the corresponding input data, but does so without knowing the particular performance benefit of doing so; it can do this, because it doesn't not need to quantify the tradeoff between obtaining disk locality against other concerns. Such ad hoc solutions will not be sufficient in multi-purpose clouds, which will exhibit higher utilization and much more diversity in consumer concerns, leading us to our proposed explicit support for soft constraints. We believe that the complexity associated with this diversity will require sound foundations, and we believe that utility functions can provide a sound foundation with sufficient flexibility.

2.2 The importance of soft constraints

We use the term "soft constraint" in relation to resource requests to refer to an indication that certain resources would more effectively serve than others, but are not strictly necessary. They differ from "hard constraints", which must be satisfied, rather than just being more desirable.

When such non-mandatory preferences exist, soft constraints can can inform scheduler decisions so that

1. the job receives *preferred* resources, if they are available;
2. the job is assigned less preferred, but nevertheless usable, resources whenever possible, which can reduce job completion delay (by reducing scheduling delay) and increase resource utilization (by not leaving usable resources idle).

There are many situations where soft constraints fit. One of the primary examples is locality, such as the disk locality that is hard-coded into systems like Hadoop. But, input data locality is not mandatory to successful execution, as evidenced by its absence from some environments using map-reduce [2], making it an appropriate soft constraint rather than hard.

A similar example is compute-compute coupling, especially in tightly coupled, inter-process latency sensitive HPC applications, where increased variability in roundtrip time between tightly coupled tasks causes synchronization delays. For example, an n-body simulation may benefit from running all of its processes on a large multi-core machine rather than across a collection of distinct servers communicating over a cluster network. If such n-body simulations are to co-exist with other workloads on the same shared infrastructure, specifying their *preference* for proximal co-location as soft constraints would be valuable.

Specialized hardware accelerators (e.g., GPUs) can also be highly desirable for a given application, but not necessarily required if emulation libraries exist. Evidence supports increased heterogeneity of cluster resources, both incidental by virtue of incremental upgrades, and deliberate heterogeneity to introduce specialized hardware best suited for certain types of workloads. For best return on investment, each platform type should be multiplexed among all workloads running on the cluster that can take advantage of it. Common examples include chipset features like SSE4 vectorized instruction sets, GPUs, and specific kernel versions. Many of these are already known to be requested as hard constraints in modern clusters [13]. Often, however, we believe that their specification as a soft constraint is sufficient and more appropriate.

3. SOFT CONSTRAINT SPECIFICATION

Specifying soft constraints is not trivial. It could be represented as a hard constraint, explicitly requiring placement on preferred machines, but that loses the advantages of the tradeoff options mentioned in section 2. We propose and introduce a method of specifying soft constraints that has the following qualitative properties:

1. expressivity – ability to express a vast array of common resource preferences with corresponding fallback options
2. composability – the individual components express simple ideas, and they can be naturally combined to form complex requests.
3. simplicity – the structure of the resulting representation is relatively simple, both for humans and for automatic generation tools.
4. backward compatibility – the same mechanism can be used to represent hard constraints or specify complete absence of constraints.

An interface that fully describes soft constraints must be able to express at least three things:

- the value associated with a specific subset of resources
- bounds on value in consideration of budget constraints
- tradeoffs between different resource subsets

We believe that all of these can be expressed via utility functions that associate a utility (i.e., value) with specific cluster resource allocations. Whereas utility functions in previous cluster schedulers were either as a function of the *quantity* of resources allocated or as a function of time [3, 10], alsched uses utility functions that are a function of specific cluster resource *subsets* (as opposed to others). Note that such utility functions also retain the ability to express a mapping of resource quantity to utility.

3.1 Equivalence classes

With the heterogeneity and dynamicity of shared resources on the rise [13, 12], 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 not all workloads care about location and machine characteristics equally. Evidence from available trace analyses [11, 12] suggests that placement constraints can range from non existent to unachievable, even in the empty cluster. In short, the logical partitioning of the cluster must be done from the perspective of the workload itself.

To address this, alsched applies the notion of *equivalence classes*. Each resource consumer can group resources into equivalence classes, based on its particular concerns, such that the units within each equivalence class *are* fungible. This simplifies the expression of soft constraints specified as utility functions. Such resource grouping can range from per-machine classes on one extreme to having the entire cluster as a single class on the other. Given this mapping, all subsequent primitives are defined over a (potentially much reduced) set of equivalence classes.

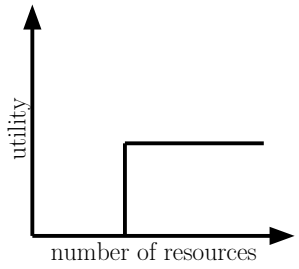

Figure 1: n Choose k primitive: utility function associating utility u with $\geq k$ resources allocated from the specified equivalence class.

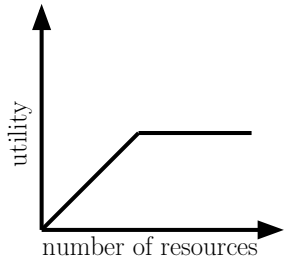

Figure 2: Linear "n Choose k" primitive: piece-wise linear utility function associating utility u with $\geq k$ resources allocated from the specified equivalence class, with a linear aggregation with $< k$.

3.2 Primitives

Resource consumers want have the ability to specify utility over both the quantity and the type of resources allocated. The equivalence classes provide ability to dynamically and logically paritition the cluster into "types", and we define primitives to map the quantity of resources chosen from a given equivalence class to utility. Then, a composition of primitives, defined with the operators introduced in subsection 3.3, aggregates the utility across potentially many equivalence classes.

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 class. 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 class and a number (k) of resources from that class to a utility value. Thus, given an assignment of resources across all equivalence classes, 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 class. Pictorially, the utility function encoded by the nCk primitive is shown in Figure 1, while the linear "n Choose k" in Figure 2 allows for linear aggregation of resources from the specified equivalence class 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.

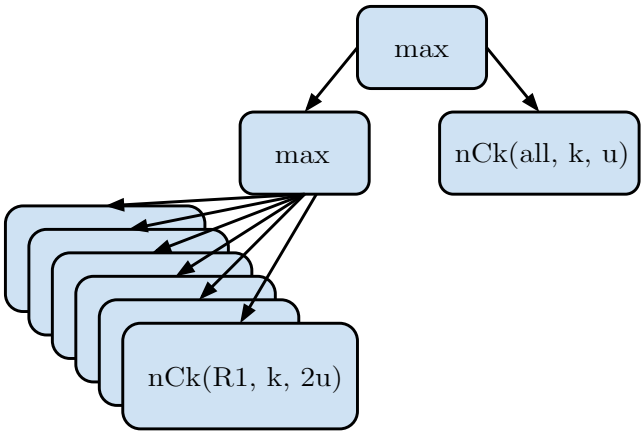

Figure 3: An example utility function that encodes a preference for colocating *k* tasks on the same rack with the fallback of scheduling these *k* tasks anywhere on the cluster.

3.3 Operators

While the primitives allow association of numerical utility with the quantity of resources chosen from a given equivalence class, the full expressiveness of alsched's utility function specification mechanism comes from the way they are composed. To compose these primitives, we introduce operators with intuitive meanings: Min, Max, Sum, Scale, and Barrier. Each of these operators take numerical utility values as input, and output 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.

3.4 Example algebraic expressions

To illustrate the expressive power of alsched's primitives and operators, we present several examples we expect to be common in modern clusters. We start with a simple locality constraint specified over a set of racks, falling back to running the same number of tasks anywhere on the cluster (Figure 3). The leaves in the left branch represent utility $2u$ assigned to the choice of k machines per rack on some rack from the enumerated list. The result of this is aggregated with a max operator and compared against the utility possible from the assignment of k tasks anywhere in the cluster. Note that if there does not exist a single rack such that k tasks can be all allocated on that rack, the left branch will evaluate to zero. If these k tasks can be accomodated somewhere on the cluster, the right branch will evaluate to u, causing the result of the whole expression to be either $2u$ if a rack is found or u if not. Note that the max operator will not distinguish betwen the assignment of a single rack, or all machines on all racks. From the perspective of a resource consumer looking for a single rack, these are equivalent: the consumer can simply choose not to use the extra assigned resources. It is the job of the cluster scheduler to avoid assigning resources that do not improve the utility value.

The second example is an anti-affinity constraint with a soft fallback. In Figure 4, the user agent is asking to be spread across racks r1 and r2, with k resources allocated on each, OR spread across racks r3 and r4, with k resources allocated on each, AND

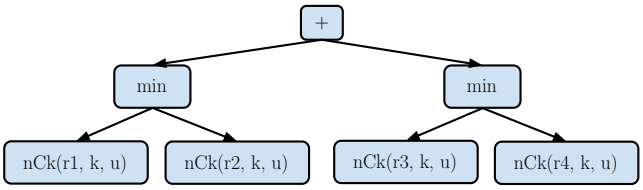

Figure 4: Utility function encoding an anti-affinity with a soft fallback to another set of racks.

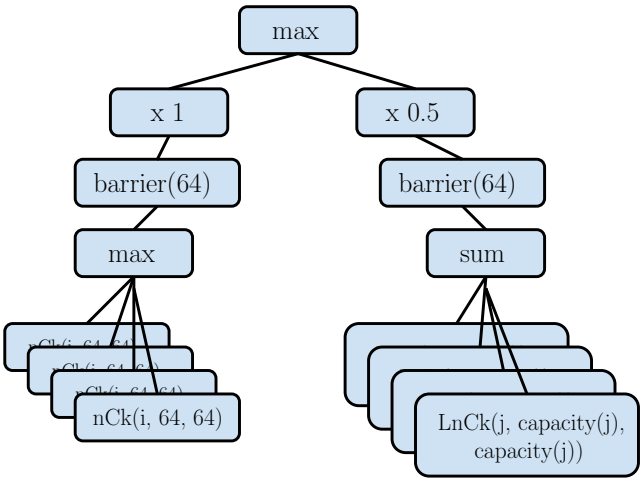

Figure 5: Utility function encoding a soft locality constraint.

ideally spread across all four. Failure to allocate at least k machines in either branch does not result in a failure. We note that we're not aware of any other constraint specification mechanism that has the expressivity to describe such a resource request.

In a third example, Figure 5 is specifying a soft locality constraint. This example is used in the simulation in Section 5. It can also be succinctly encoded for recursive automatic processing as a prefix expression as follows: (max (*1 (bar 64 (max (nck(i,64,64))))(*0.5 (bar 64 (LnCk(j,cap(j), cap(j))))))

In this example, we have 64 tasks for n-body simulation, which we'd like to colocate on the same 64 core machine (left branch). Failing that we are OK with running these tasks on the rest of the cluster (right branch). Note that the barrier ensures that we have at least 64 tasks allocated, the scale ensures that preference be given to the left branch, if it is satisfied, and the max at the top ultimately picks the winning branch. If the 64-core machine is available, choosing it will maximize utility for this utility function.

4. ALSCHED DESIGN

To evaluate our proposed method of composable utility functions, we implemented **alsched** – a system for algebraic scheduling that allocates resources in accordance with the composable utility functions submitted as resource requests. Figure 6 illustrates the overall system model and the model of interaction between resource consumers and the alsched scheduler. The alsched scheduler manages a set of resources and exposes their characteristics to allow resource consumers to determine equivalence classes appropriate to their interests. Resource consumers specify their resource requests in the form of algebraic utility expressions. Submission of these utility functions is envisioned to occur at the following points in time:

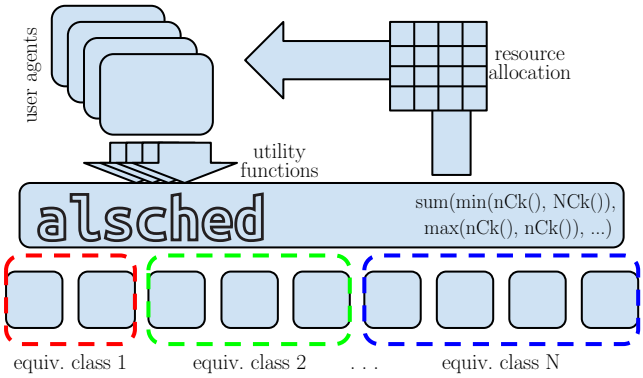

Figure 6: alsched System Model

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 itself could fire 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 a new utility function to be scheduled. Having thus accumulated a non-empty set of utility functions, the scheduler can either optimally or greedily perform the assignment of resources in a way that maximizes the overall utility of the cluster. 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 class j to resource consumer i. Each individual agent, upon receipt of its corresponding allocation vector $D[i]$ has the ability to evaluate the utility value of this assignment, based on the provided utility function. If the value is zero, it simply remains in the pending queue to be scheduled again, potentially with a new/updated utility function.

We setup the problem of resource allocation as an optimization problem, taking advantage of the fact that the primitives and the operators were chosen such that the composed result has the monotonicity property. This allows us to use branch and bound as one of the algorithms to solve the assignment. In the simulation described in Section 5, we use greedy assignment to speed up the performance of the scheduler proper, while still achieving simulated cluster throughput improvements in the case when soft constraints are specified.

5. PRELIMINARY EVALUATION

We have conducted a series of experiments with the primary focus of motivating the usefulness of soft constraints and their effect on resource scheduling effectiveness. The results also serve as evidence that alsched's utility function design can work as described. We observe that utility functions do indeed guide the scheduler to allocate appropriate resources to the requester, in an effort to maximize the overall utility of the cluster.

5.1 Simulator

We developed a simulator capable of playing through a set of jobs of the following 3 types: NBODY, HADOOP-PI, and LOCAL. The NBODY type was chosen to represent a tightly coupled compute-bound workload that can run across many servers in a distributed fashion, but gains an advantage in performance when all of its resources are colocated on a single server. Thus, the NBODY type is a workload that can benefit from soft constraints. The HADOOP-PI type was based on Hadoop's Monte Carlo π computation. It is embarrassingly parallel, has no locality constraints, and has throughput linear in the number of resources assigned to it. The LOCAL type represents an application that must run in a single address space and, thus, requires all resources to be allocated on a single server.

The simulator admits a set of synthetically generated jobs. Newly arrived jobs become schedulable as soon as the simulation time catches up with their start time (jobs may be scheduled into the future). As soon as the scheduler allocates resources to a schedulable job, that job is transitioned to the running state and eventually retires upon reaching its finish time. The simulator continues running for as long as there are schedulable tasks in the queue and stops when the queue is empty.

5.2 Experimental Setup

The simulator is driven by a set of configuration parameters spanning 3 major categories: job, cluster, and scheduler configuration options. Jobs can vary in number, size, duration, and mean inter-arrival time. Though we have experimented with a range of options, the results presented in this section used a fixed value of 100 time units for the task duration, with all tasks arriving at time $t = 0$. Cluster parameters include the total number of machines and their breakdown into the set of large 32-core machines, which were scarce and contended, and regular 8-core machines, which were available in greater quantities. The ratio between these two classes of machines was varied to illustrate how it affects scheduler behavior. The scheduler proper can be configured to use different algorithms (for placement computation purposes), including branch and bound and greedy search, and any of three constraint handling policies. The "Soft" constraint option allocates resources in accordance with the proposed alsched design, respecting soft constraints and giving priority to utility function branches that result in higher utility. The "Hard" policy converts soft constraints into hard constraints, treating them as an absolute requirement, and the "None" policy simply ignores any specified soft constraints.

For the purposes of demonstrating the utility of soft constraints, the default configuration was a cluster of 88 nodes, with 8 32-core machines and 80 8-core machines. With this configuration, the total capacity of the 32-core machines is 256 cores, and the total capacity for 8-core machines is 640, providing a capacity ratio of 4:10. For each simulation run, we generated a total of 500 jobs all arriving at time 0. The number of tasks for each job was binomially distributed with $p = \frac{1}{8}$ and $n = 64$. The runtime duration of each job depended on whether it received a locality speedup from being scheduled on one of the desired machines, i.e., one of the machines with enough cores to accommodate all tasks, in the cases of NBODY and LOCAL. The speedup factor is also configurable, and we varied it from 1.2 to a factor of 10, correspondingly reducing the job duration enjoying this speedup.

5.3 Preliminary Results and Discussion

Figure 7 plots the amount of time (which we call total runtime) it takes the cluster to work through a given queue of NBODY type jobs, for the three different constraint policies at different speedup factor (f) values. We note that the soft constraint policy outperforms ignoring constraints("None") for all speedup factors and is better than treating them as hard constraints until the speedup factor is dialed up to about 10.

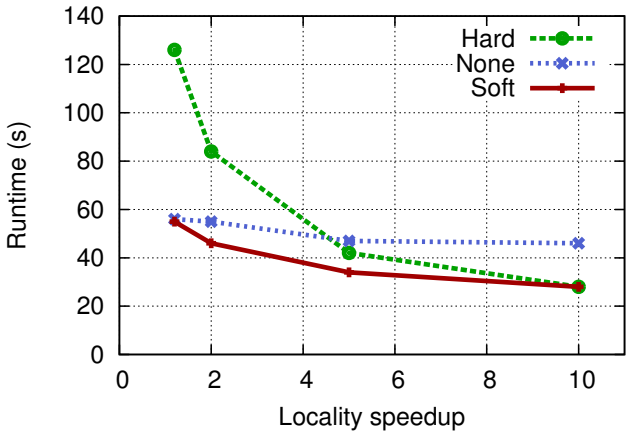

Figure 7: Total runtime with "Soft", "Hard", and "None" scheduling policies over the range of speedup factors gained through locality. Large:small machine capacity ratio : 4:10 (non-scarce)

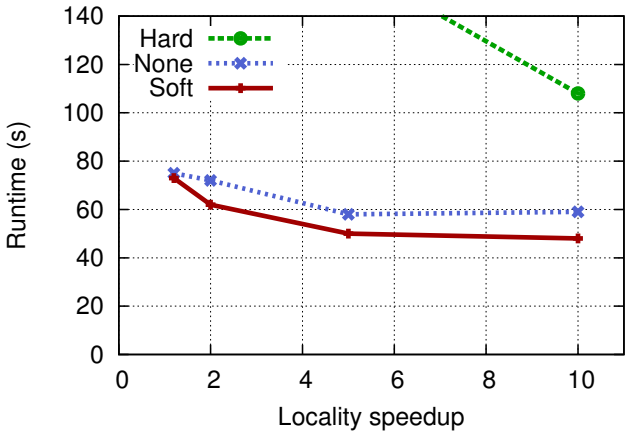

Figure 8: Total simulated runtime with large:small capacity ratio 1:10 , zoomed in on "None" and "Soft" (full graph shown in Figure 9).

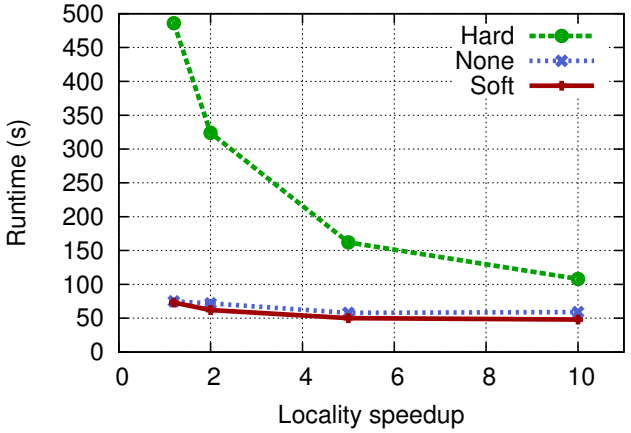

Figure 9: Total runtime over a range of speedup factors for "soft", "hard", and "none" scheduling policies. Large:small machine capacity ratio = 1:10

"Hard" and "None" can be viewed as two endpoints of a continuum. At one extreme, the scheduler *never* waits for more suitable resources and schedules as soon as it finds spare capacity. At the other extreme, the hard constraint policy *always* causes the job to wait for resources that it has specified a preference for. Often the best solution fits between these two end points. The "Soft" policy pushes the solver in the direction of preferred resources, which is reflected in a higher throughput gained. Yet, there does exist an inflection point at which it becomes better to wait for desired resources than to take advantage of immediate availability of the fallback machines. The intuition is that this inflection point represents the point where the difference in scheduling latency won by exploiting the fallback is insufficient to cover the temporal cost paid for running on "slower" machines.

To illustrate that tradeoff further, Figures 8 and 9 show the same experiment with fewer large machines, namely with a reduced capacity ratio of 1:10 (instead of the previous 4:10). In this configuration, the cost of waiting for a 32-core machine becomes higher due to contention for desired resources, and running on secondary or tertiary choices becomes more preferable with higher locality speedups available. Indeed, the inflection point, where soft constraints no longer help, is pushed further out, beyond $x = 10$. The "Soft" constraint policy still outperforms "None", albeit by a smaller margin for most settings. This is expected, since the fast resources are fewer. The gap between "Soft" and "None" is a function of desirable resource availability or, equivalently, the probability with which the more desired branch can be chosen by the scheduler. The gap between "Soft" and "Hard" is exacerbated by the scheduling delay suffered by the "Hard" policy as it waits for scarce desirable resources.

Thus, we've identified several important factors that influence the scheduler's choice of placement policy: a speedup gained by running on preferred resources, scarcity of and contention for preferred resources, and job duration. These three factors define a tradeoff space the optimal scheduler should explore and, to be optimal, it needs to be informed of or able to predict them. alsched's utility function design provides such information for the scheduler, allowing it to find good solutions.

In a second set of experiments we tested 4 workloads: a homogeneous workload for each job type, and a heterogeneous mix of all job types. These results are shown in Table 5.3. Note that the default job length in this experiment was 10 time units. The HADOOP_PI and LOCAL workloads contain no soft constraints and serve as a control to show that all policies perform the same in these cases. For the NBODY workload, in which every job has soft constraints, the "Soft" policy performed better than the other two. In the mixed workload the advantage of the "Soft" policy was diminished, but it performed at least as well as "Hard", while outperforming "None" by more than a factor of 2.

6. CHALLENGES

Soft constraints are important, and we believe that alsched's approach of composable utility functions is promising, but substantial research remains. The list of interesting challenges includes automation of utility function construction by execution frameworks (e.g., Hadoop), scheduler stability in presence of imperfect utility functions, and comparison of utility functions amongst consumers.

	PI	NBODY	LOCAL	MIXED
Soft	22	12	33	22
Hard	22	18	33	22
None	22	17	33	51

Table 1: Simulated running time for different combinations of placement policy and job type. Each simulation started with a queue of 100 jobs arriving simultaneously. The cluster was configured with 8 32-core servers, and 64 8-core servers. The numbers in the chart indicate the total time to finish all jobs.

We designed alsched's utility function scheme with the recognition that a typical user is unlikely to construct one by hand. The goal is to make it possible for the power users, leaving the maximal flexibility and expressivity available to them, while letting commonly-used frameworks automatically generate utility functions based on a framework's knowledge of current task and data location and higher-level user input.

For the utility functions that are provided, especially given the uncertainty in job duration, we anticipate that the tradeoffs and the costs associated with them often will not be precisely captured. The scheduler must be designed to be resilient in the face of underspecified or imprecise utility functions. One possibility is to use the utility function mechanism to outline a high-level picture of placement preferences. When the solution feasibility space is explored by the scheduler, intermediate solutions could be offered to the user agent or her framework for evaluation. This would be akin to resource offers in Mesos [4].

We suspect that constraints are more likely to be specified by longer running jobs, for which resource assignment changes may be needed over time to maximize utility. But, change is rarely cost-free, and too many or poorly chosen changes can result in lower utility rather than higher. Good algorithms and interfaces for identifying changes that appropriately consider their consumer-/context-specific costs and benefits will be needed.

7. CONCLUSION

Soft constraints will be important to achieving the efficiency promise of cloud computing, given the diverse demands and characteristics of applications that will share consolidated heterogeneous infrastructures. Appropriately formulated utility functions, exploiting primitives like "n Choose k" across consumer-specified sets, provide a flexible and effective way for consumers to specify their particular soft constraints. Given such utility functions, a cluster scheduler can serve consumer needs more effectively than when preferences are treated as hard constraints or not considered at all.

Much work remains in fully demonstrating the vision outlined in this paper and evaluating the tradeoffs involved in realizing it. We are in the process of porting alsched into a cluster scheduler [7] used in several cloud computing testbeds, as well as porting consumer frameworks to specify utility functions appropriately, so as to evaluate its effectiveness in practice. We are also expanding the simulator to allow for experiments with more complex and large-scale scenarios. Despite the research still needed, however, we believe that the approach outlined herein is a very promising way to address many of the complex scheduling challenges faced by future cloud infrastructures.

8. ACKNOWLEDGMENTS

Some of the ideas presented here benefited from discussions with Matei Zaharia and our other collaborators from UC-Berkeley. 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) and by an NSERC Postgraduate Fellowship.

9. REFERENCES

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

[2] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proc. of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 12–12. USENIX Association, 2011.

[3] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. 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.

[4] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. 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.

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

[6] T. 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.

[7] M. Kozuch, M. Ryan, R. Gass, S. Schlosser, D. O'Hallaron, J. Cipar, E. Krevat, J. López, M. Stroucken, and G. Ganger. Tashi: location-aware cluster management. In *Proc. of the 1st Workshop on Automated Control for Datacenters and Clouds*, 2009.

[8] K. Lai. Markets are dead, long live markets. *SIGecom Exch.*, 5(4):1–10, July 2005.

[9] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B. A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, 1(3):169–182, Aug. 2005.

[10] C. B. Lee and A. 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.

[11] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. 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.

[12] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. 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.

[13] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. 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.

[14] I. Stoica, H. Abdel-wahab, and A. 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.

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