# Service contracts and aggregate utility functions

Alvin AuYoung, Laura Grit, Janet Wiener and John Wilkes
UCSD, Duke University and HP Laboratories
alvina@cs.ucsd.edu, grit@cs.duke.edu, janet.wiener@hp.com, john.wilkes@hp.com

*Utility functions are used by clients of a service to communicate the value of a piece of work and other QoS aspects such as its timely completion. However, utility functions on individual work items do not capture how important it is to complete all or part of a batch of items; for this purpose, a higher-level construct is required. We propose a multi-job* aggregate-utility function*, and show how a service provider that executes jobs on rented resources can use it to drive admission control and job scheduling decisions. Using a profit-seeking approach to its policies, we find that the service provider can cope gracefully with client overload and varying resource availability. The result is significantly greater value delivered to clients, and higher profit (net value) generated for the service provider.*

## 1 Introduction

We have a colleague who often runs 100,000 jobs over a weekend on a shared compute cluster in order to perform an experiment. Each job takes a few minutes to run and produces one data point on a graph. The graph is nearly useless if too few data points have been obtained by Monday morning, but completing 90% is almost as good as completing all of them. No particular job is more important than any other – it is the aggregate set of results that counts.

Computer-graphics film animators often compete with each other for access to a compute farm on which they run multi-hour rendering jobs overnight [3]. For any particular animator, getting a particular image back the following morning has some benefit, and having more images rendered is better – but sometimes the majority of the sequence needs to complete for any of it to be useful. Some frames are considerably more expensive to render than others. No particular frame is more important than its peers – it is the overall effect that matters.

Outsourced business services often have service level agreements (SLAs) or contracts that include penalties for poor performance: if the response time is too high, for too many transactions, the service provider will earn less, and may even have to pay out more than it takes in. No individual transaction is any more important than any other – the percentage of transactions that violate the bounds is what is important.

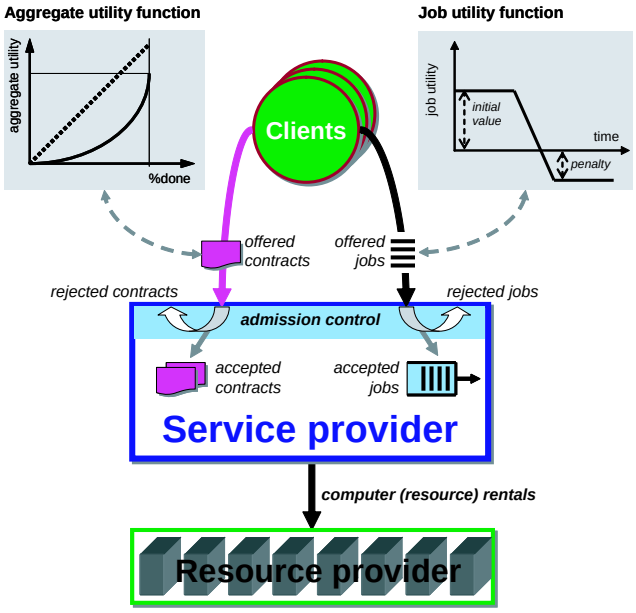



**Figure 1. Problem overview.** *The relationship between clients, service providers, and resource providers.*

Simple per-job or per-work-item information does not capture the true intent of the client in these examples, leaving the service provider to do the best it can, but risking unhappy clients, under-utilized services, or both. What is needed is additional control that can express the client's desires while not unduly constraining the service provider. This paper presents such a control, and evaluates its behavior.

We focus our study by considering the concrete example of a *job-execution service provider* that runs batch jobs on behalf of its clients, who can be commercial clients, scientific partners, or other entities.

### 1.1 Contributions

The primary contributions of this paper are to:

- introduce *aggregate utility functions*, which are used in contracts to let clients specify the overall value of completing a set of work, in addition to the values of individual work items;
- present algorithms that allow a service provider to make both per-contract and per-job admission-control and scheduling decisions that take such aggregate utility functions into account; and
- evaluate these algorithms by means of a simulation study, in the context of a service provider that obtains resources from an external source.

Our evaluation covers a range of operating conditions: load, resource cost and quantity variability, per-job utility function shape, and aggregate utility function shape. Our experiments show that the new algorithms consistently extract higher utility for clients and higher profit rates for service providers than previous approaches.

The next section introduces our model of services and the contracts they support; section 3 describes our job execution service in greater detail; section 4 describes the resource provider service it uses. The evaluation portion of the paper starts with a description of our setup in section 5 and is followed by our results in section 6. A survey of related work and our conclusions close out the paper.

## 2 Services and contracts

*Service-oriented computing* has arrived in enterprise computer systems [19]; the Grid can be viewed as its non-commercial counterpart, and has similar momentum behind it [13]. In such environments, large-scale functions such as business processes are composed from separable, loosely-coupled services that can be reused and shared. Tools and techniques that increase the effectiveness of these environments have become important.

One immediate opportunity this environment offers is to outsource work to specialized services, such as a job-execution service. This outsourcing has many benefits: clients avoid the hassles and expense of maintaining their own data center and provisioning it for their peak load; clients benefit from the service provider's economies of scale, expertise, and management; and multiple clients can share a common infrastructure and (sometimes) expensive software licenses, resulting in lower costs.

Like many others working in this space, we have found that charging for services clarifies the value that such services provide, and brings a helpful precision to the notion of goodness, utility, and benefit.[1]

The *profitability* of a service is a direct measure of the amount of value the service is adding to its environment. It is simply the difference between the cost of running the service and what its clients and sponsors will pay to do so. Although profit is clearly a useful measure in the commercial sphere, we believe that it offers a helpful measure of added value in non-commercial settings, too. In both cases, it is important to have a clear measure of how best to use limited resources, whether they be computer equipment, people's time, or funds. Some measure of goodness must exist if alternate designs are to be compared, and mapping this measure onto a numerical scale allows quantitative reasoning and computer-based optimizations [15].

Real money has the advantage of fungibility and external comparability – if you do not like a service, you can spend the money on buying it elsewhere, or on building it yourself – but any relatively scarce "currency" has many of the same benefits, as long as there is a way of mapping its quantity back to a metric of interest, including all the QoS "-ilities" that matter in a particular circumstance. As we will show, economic mechanisms can drive the selection of algorithms to use in a service provider in productive ways.

Payment for a service can come from many sources: from its clients, from third parties such as advertisers, or from sponsors such as an IT department or a government funding agency. In this study, we use a pay-per-use model, funded by the clients.

### 2.1 Contracts

In a service-oriented world, clients need control over their service provider's behavior, and service providers must be able to constrain the behavior of their clients. This mutual control is provided by means of *service level agreements*, or *contracts*, which specify the service to be provided, its quality and quantity levels (e.g., the load that the client can impose), price, and penalties for non-compliance.

Too much specificity in a contract may prevent helpful optimizations behind the scenes; too little leaves the the service provider to second-guess the intentions and desires of its clients, which exposes the clients to the risk of being surprised, disappointed, or both.[2]

For our job-execution service example, each of our clients negotiates a contract with the service provider to run a single sequence of jobs. The client binds itself too, by including a description of the contract's workload in sufficient detail to allow its aggregate load and value to be estimated, but not the precise timings of when jobs will arrive, or their

[1] We distinguish between charging for services and pricing them. Market-based mechanisms such as auctions offer one way to price goods and services, but there are plenty of others. We concentrate here on what to do with the information that prices provide.

[2] Not everything needs to be explicitly specified in a contract: anything for which there is little risk of misunderstanding can safely be omitted. Ascertaining what is mutually understood is itself an interesting problem.

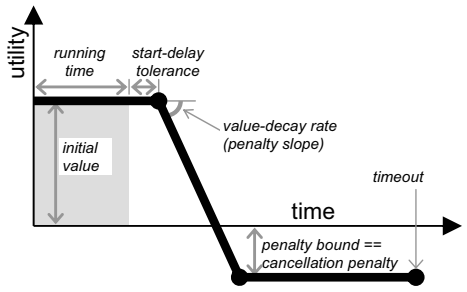



**Figure 2. Per-job utility functions.** *How much value a job delivers as a function of when it is completed.*

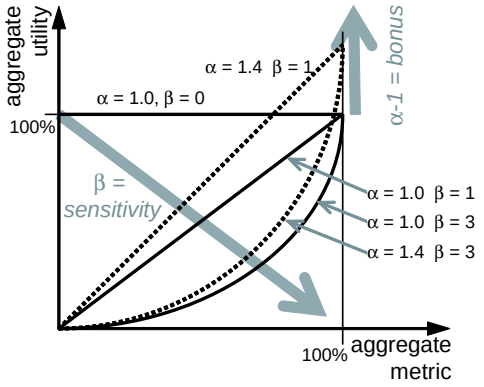



**Figure 3. Aggregate utility functions.** *Some representative functions, showing the effects of changing $\alpha$ and $\beta$ on the total pay-out for a contract.*

individual sizes or values. This description includes estimates of the number of jobs, their sizes, arrival rates, and utility functions, in the form of distributions (in our case, the distributions used by our client workload-generators).

We model well-behaved clients that submit jobs that conform to the contracts they negotiate; coping with malicious clients is outside the scope of this paper. For simplicity of exposition, each job demands only one processor; handling multi-node, moldable or reshapable jobs is a relatively straightforward extension.

## 2.2 Job utility functions

Each job has an associated time-varying *utility function* that expresses the maximum price that the client is willing to pay for that job to be run, and how this price decreases with elapsed time (see Figure 2). We follow previous work in this field [7, 14, 21], and use a simple three-part shape for the per-job utility function; these utility functions start at some constant value and continue there for a while; decline at some constant rate (to reflect increasing disappointment in a late result), and eventually reach a maximum negative penalty. Finally, the function reaches a timeout, indicating that the client is no longer interested in the job's output.

We equate the value of a job with the maximum price the client is willing to pay, and for simplicity (and greater focus on our problem) ignore price-setting mechanisms such as auctions. Our clients always pay their jobs' true value.[3]

Once the job execution service provider accepts a job, it will either run it and deliver its results, or it will cancel it. If the job is not completed by its timeout, it is always cancelled. The job value to the client equals either the job's utility function value at the moment that the job completes and returns its results, or the maximum penalty if the job is cancelled.

[3]Prices are likely to be lower with multiple, competing service providers, which will in turn increase the incentive for such service providers to operate efficiently and competitively – and their need for algorithms such as the ones we discuss here.

## 2.3 Aggregate utility functions

In the absence of a higher-level contract, the service provider is free to cherry-pick jobs, and accept only the most profitable ones, leaving the client at risk of not getting its less-profitable work done.

To prevent this behavior, the client needs to constrain the service provider somehow. As the examples in the introduction show, simple per-job metrics will not work, and imposing binary constraints of the form "you must finish all of these" would be sub-optimal in the presence of competition from other clients that may not be known at the time the contract is written.

For our solution, we build on top of per-job utility functions and choose the following: the overall payment for a contract is the sum of the per-job prices multiplied by the value of an *aggregate utility function*, which is a function of an aggregate metric measured across the entire contract. This function allows the client to express near-arbitrary consequences for different aggregate behaviors in a simple way. We believe that this mechanism is simple, powerful, easy to communicate, captures important client concerns, and is easy for the service provider to interpret.

The aggregate utility function could be of nearly arbitrary shape. To explore a range of behaviors, we picked a family of functions that can be generated using only two parameters: $aggregate\_utility = \alpha x^{\beta}$ for an aggregate metric value $x$ in the range 0–1 (see Figure 3). We call $(aggregate\_utility - 1)$ a "bonus" when it is positive, and a "penalty" when negative.

The parameter $\beta$ is a measure of the client's sensitivity to the aggregate metric: when $\beta = 0$, the client is indifferent to its value; when $\beta = 1$, the client is moderately sensitive (the relationship is linear), and for higher values of $\beta$, the client is increasingly sensitive. When the aggregate metric is the fraction of jobs completed, then as $\beta$ increases, the client is

expressing increasing concern about completing all of the jobs; for $\beta < 1$, as $\beta$ approaches 0, the client is expressing increasing indifference to having the last few jobs run.

The parameter $\alpha$ describes the potential upside to the service provider of good performance: for example, with $\alpha = 1.4$ and $\beta = 1$, the client is offering a bonus of 40% of the sum of individual job utility values for completion of all the jobs in a sequence (the tallest straight line in Figure 3).

The overall pay-out for a contract is calculated at its end; we assume that payment can be deferred until then, and any disputes can be arbitrated by a third-party auditor [2, 4].

Here, we use the fraction-of-jobs-completed as the aggregate metric, but it could as easily be *any* such metric, such as the average job completion time, the average start-time delay, or even the correctness of the results.

Composite utility functions could certainly be constructed using more than one aggregate metric. Essentially, they become objective functions for the service provider, guiding its tradeoffs along different operating dimensions. Using such functions might be an interesting way to augment the penalty clauses that are traditionally used in SLAs to handle QoS violations for properties such as availability, reliability, correctness, timeliness, and security – but it remains future work.

We also believe that aggregate utility functions that span multiple contracts would be a powerful tool to capture concerns about overall customer satisfaction and most-favored customers; such functions are also potential future work.

### 2.4 Using contracts in the service provider

Faced with a proposed contract from a client, a service provider has to decide whether to accept or refuse it. Once a contract has been accepted, the service provider is bound to it: it cannot be cancelled, although it can effectively be abandoned. Payment is determined by the combination of jobs completed and the aggregate utility function.

Even in the absence of penalties for refusing contracts, the service provider still faces a tricky question when a new client contract arrives: is accepting the new one likely to give it more profit than completing an already-accepted one that it might have to abandon, or even some possible future one? Remember that the contract details provide only estimates of future client behavior – the details of exactly which jobs will arrive when are unknown, for both the new and the existing contracts.

The number of resources available to the service provider may fluctuate with time, affecting which contracts it can service profitably. The desirability of abandoning an existing contract is affected by both the likely cancellation penalties for its jobs, and by what fraction of the achievable aggregate-level benefits have been achieved from the already-completed jobs: if it has nearly completed a contract with a high $\beta$ value, it may well be worth completing it, because much of the payout will result from only a little more work. All these factors complicate the service provider's decision.

A similar problem occurs when the client submits a job: the service provider has to decide whether to accept it or not, bearing in mind the likely profitability of the job by itself, its impact on other work that it has already agreed to do, and the effect it might have on the aggregate utility function for the contract the job is associated with – or even other contracts, if those jobs have to be cancelled to make way for this one.

The next section describes some of the algorithms we use to solve these problems.

## 3 Job execution service

The primary metric we use to evaluate the job execution service is the *profit-rate* it achieves: the difference between its income and expenditures per unit time. Income corresponds to the utility (value) it delivers to its clients, as measured by what they pay; expenditures are its costs to rent processors on which to run the jobs. In turn, client value is specified by the combination of a contract's per-job utilities and the client's aggregate utility function.

We are also interested in the total client utility achieved, which we equate with the total value (utility) the clients pay – i.e., the service provider's revenue.

The job-execution service provider runs two types of admission control algorithms: one for client contracts and one for individual jobs. It also has a scheduler that decides when to run jobs. We discuss these algorithms in the remainder of this section.

### 3.1 Contract admission control

The *contract admission control algorithm* determines which client job-sequences to accept. Its purpose is to avoid long-term service over-commitments, and to establish a binding contract between the service provider and its client. The algorithm is run whenever a new contract arrives. It first runs a feasibility check to determine if it *can* accept the contract (i.e., it will be able to get enough resources to do so), and then a profitability check to see if it *should* (i.e., if its profitability is likely to increase if the contract is accepted). If the contract passes both tests, it is accepted; if not, it is declined, and the client seeks service elsewhere. There is no penalty for refusing a contract, but once accepted, it is mutually binding on both parties.

The contract-feasibility check is selected from the following policies:

1. *contract-load=oblivious*: always accepts contracts.

2. *contract-load=average*: accepts a contract only if its average load plus the existing average load is within

the predicted resource availability for the contract's duration.

3. *contract-load=conservative*: like average, but uses load estimates that are 2 standard deviations above the average, to provide some resilience to time-varying loads.

4. *contract-load=preempt-conservative*: accepts a contract if it passes the *contract-load=conservative* admission test, possibly by cancelling an overlapping contract that would generate less total revenue.

5. *contract-load=high-value-only*: accepts a contract if its expected value per hour exceeds a threshold. Setting the threshold requires knowing the expected value per hour of future contracts; however, running this algorithm provides a useful upper bound on profit.

If the contract is feasible, its profitability is then checked, using one of the following tests:

1. *contract-cost=any*: the cost to rent resources to execute the contract is ignored; no contract is rejected for this reason.

2. *contract-cost=variable*: profitability predictions are calculated separately for each different cost period and added; only contracts that increase the overall profit-rate are accepted.

In theory, multi-round negotiations could occur at the time a contract is offered [12]. For simplicity, we just consider a contract once, and accept or reject it as it stands: no attempt is made to adjust the contract to make it more acceptable to either side.

Once a contract is accepted, clients can submit jobs against it.

### 3.2 Job admission control

To avoid short-term overload, the job execution service provider also runs a *job admission-control* algorithm when a new job arrives, which decides whether it should accept individual jobs. Rejected jobs are not further considered, although they do affect the aggregate utility metric. Note that in our test workloads, some jobs are individually unprofitable: even if there were no other jobs in the system, the cost of executing them would exceed the job's utility.

If the admission control algorithm accepts a job, it is placed into a work queue, from which it is selected to be run at some future time by a *job scheduler* (section 3.3).

In the absence of aggregate utility functions, the job admission decision is made by comparing the profit rate of a tentative new schedule that includes the new job against the existing schedule that does not.[4] If the new profit rate is higher, the job is accepted.

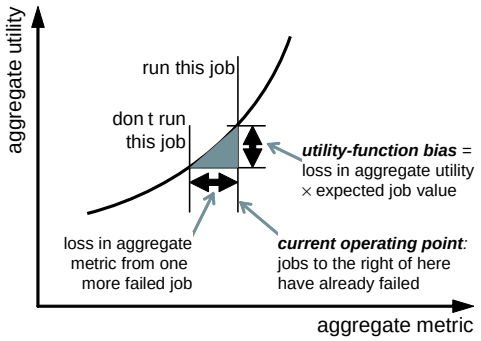



**Figure 4. Calculating the effective utility bias.** *The bias is calculated from the loss in aggregate utility of not running a job, starting at the current operating point.*

Aggregate utility functions complicate job admission control: it may be more profitable to run an individually-unprofitable job than to reject it. A useful way to think about this scenario is to consider the cost to the service provider of *not* running a particular job. The lost aggregate utility may be larger than the cost of running a non-profitable job. Our solution is to make the job appear to be sufficiently profitable for it to be accepted and run.

We capture the increased desirability of a job by constructing an *effective* job utility function for it that includes a *bias* to the job's utility, and use it in admission and scheduling decisions, rather than the original utility function.

Figure 4 shows how the bias is computed, and Figure 5 provides the equivalent pseudo-code. We first determine the current *operating point* (*old_fraction*), the fraction of jobs that would be finished if this job and all subsequent jobs were run to completion (i.e., 1 minus the fraction of jobs that have been rejected or cancelled so far). The bias is the drop in the aggregate utility function from completing one fewer job, multiplied by the expected (i.e., average) value of a job. It is added to all of the original job utility-function *y*-values to generate the effective utility function.

The effect of the bias is to assign much higher values to jobs that would have a large effect on the aggregate utility function if they were abandoned – for example, at an operating point near 100% for large-$\beta$ functions (highly-sensitive clients) – but not to alter the values of jobs that are at relatively insensitive portions of the aggregate utility function.

As a concrete example, suppose that the contract specifies 20 jobs with a mean job value of 15, and further suppose that the service provider has already failed to finish 2 of 12

[4] As make-span for this calculation, we use the time to first free resource; using time to last job completed gave less satisfactory results.

```
computeEffectiveUtility(
 input:
   U(t)       // original job utility function
   f(x)       // aggregate utility function
   total      // number of jobs in the sequence
   dropped    // number of jobs already abandoned
   mean_value // average for all [future] jobs

 output:
   U'(t)      // effective utility function
{
  // "old_fraction" is the position on the
  // aggregate-utility curve before this job
  // arrived - i.e., "x" in f(x)
  old_fraction = 1 - (dropped / total);

  // "new_fraction" is the new position on the
  // aggregate-utility curve if this job
  // were to be dropped
  new_fraction = 1 - ((dropped + 1) / total);

  // potential is how many jobs could complete
  potential = total - dropped;

  // bias is how far to boost the job's
  // effective value
  bias = mean_value *
         ( (potential     * f(old_fraction))
         - ((potential-1) * f(new_fraction))
         - 1 );

  // create the effective utility function
  U'(t) = U(t) + bias;
}
```

**Figure 5. Calculating effective utility.** *How effective utility is calculated for a job.*

jobs so far. When the next job arrives, the operating point *old_fraction* is 0.90, the *new_fraction* is 0.85, and *potential* is 18. Therefore, the bias is $15 \times (18 \times f(0.90) - (17 \times f(0.85) - 1)$. If the aggregate utility function $f(x) = 2x$ ($\alpha = 2, \beta = 1$), then the bias is 37.5. A more sensitive aggregate utility function $f(x) = x^3$ generates a bias of only 25.2, because the 10% of jobs that were already missed have pushed the operating point to a place where the aggregate value is heavily degraded. The same $f(x) = x^3$ correctly generates a bigger bias of 41.25 when the current operating point is 1.0 because the downside of not running even one job is so large near the 100%-complete operating point.

The jobs' effective utility functions are only used to help the service provider make decisions: they are not used for charging the client.

### 3.3 Job scheduling

Once a job is accepted, the service provider's job scheduler decides when it should be run. Since job-scheduling is NP-hard, heuristic solutions are normally used. Our approach is to use a *first-profit-rate* scheduler, which is a greedy, cost-aware scheduler derived from *first-profit* [21]. The algorithm sorts jobs by declining expected contribution to profit-rate, and selects the first job to execute from the front of that list.

The job scheduler maintains a preferred schedule of pending jobs, and attempts to execute that schedule on the resources available to it. The scheduler is invoked whenever a job arrives, a job completes, or the number of available resources changes. If the scheduler decides that a job can be run, it selects a resource from those obtained from a resource provider, and assigns the first job in the schedule to it; this procedure is repeated until the scheduler has no more jobs that can be run or no remaining available resources.

Once a job starts running, we assume it will run to completion: it will not be preempted or aborted. To avoid getting tangled up in all the issues related to managing uncertainty, we deliberately assume that the job-execution time is known in advance, and construct the schedules so that the resource provider never needs to take away resources being used by a running job.[5]

If a job's start is delayed too long, its effective value may become negative. The scheduler will then cancel the job.

## 4 Resource provider service

Much prior work has assumed that the service provider owns the machines on which it runs its service. Besides the obvious disadvantage of representing a static capital investment in a single service offering, this approach biases the decisions made by the job-execution service towards maximizing the utilization of its processor nodes, even at the cost of declining marginal utility.

We believe that there is a better way, since in a service-oriented computing world, service providers can be clients of other service providers. In particular, we model a job-execution service provider that rents compute nodes from one or more *physical resource service providers*, or just *resource providers*.[6]

Such resource rental has many benefits: the job execution service can scale up or down its capabilities as its business fluctuates; it does not have to be in the capital and operating-expense intensive business of running data centers; it can benefit from competition across multiple resource providers; and it can aggregate resources from several of them. There are disadvantages, primarily that the number and cost of resources available to the job-

[5] We realize that this is a strong assumption, but it is a conscious one, because it makes it easier to focus attention on the new results. Nevertheless, we believe that our results would be similar without this assumption – just harder to interpret, and with yet more workload parameters to set.

[6] The resources rented could be virtual machines rather than physical ones, as in Tycoon [17]. This represents a level of indirection that does not affect our story, other than to add the complexity of managing potentially dynamically-changing resource performance.

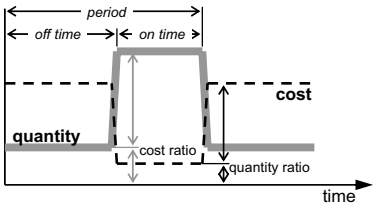



**Figure 6. The on-off model for variable resource providers.** *"On" periods correspond to times of high resource availability and relatively lower resource cost.*

execution service may fluctuate as a function of other service providers' demands on the resource provider.

Since the resource provider is not the focus of this paper, we adopt a model of its behavior that is sufficient to capture several of its salient behaviors and exercise the job-execution service provider's algorithms, while eliminating what we feel is unnecessary complexity.

Our *variable resource provider* models changing resource availability by alternating between *on* and *off* modes (see Figure 6), with more resources available in the former than the latter, and potentially different per-hour rental costs in the two modes. The special case of identical numbers and costs of resources in both on and off modes is called a *static* resource provider.

We found it helpful to construct different scenarios by considering a number of ratios between the properties of on and off periods:

- *quantity-ratio*: the number of resources in an on-period divided by the number in an off-period.
- *cost-ratio*: the cost of a resource in an on-period divided by the cost in an off-period – we expect that an excess of resources might cause the resource provider to lower its cpu-hour cost in times of plenty.
- *on-ratio*: the length of on-periods divided by the sum of on- and off-periods.

We always use equal-length on and off times (on-ratio = 0.5); the static provider has all other ratios equal to 1.0.

The resource provider offers accurate descriptions of how many resources it will have available at a specific time, using a set of tuples of the form <*start-time, duration, resource-quantity, cost*>. Prior work [21] has shown how to handle inaccurate resource estimates in a similar context to ours, so we omit that feature here.

We restrict this analysis to homogeneous processor resources; the techniques we describe can readily be generalized to handle multiple resource types.

Finally, we note that the resource provider's revenue is the same as the job-execution service provider's cost.

**Table 1. Default simulator parameter-settings.** *Our currency units are called florins. The notation* distribution(x, y) *means a distribution of the given type with a mean of* x *and a standard deviation of* y.

| *Parameter* | *Default value* |
|---|---|
| Simulation length | 1000 hours |
| Runs per data point | 10 |
| Client inter-arrival time | exponential(1.0) hours |
| Client (contract) duration | Gamma(100.0, 25.0) hours |
| Aggregate-utility $\alpha$, $\beta$ | 1.0, 0.0 |
| Job inter-arrival time | exponential(0.15) hours |
| Job length | Gamma(1.0, 0.25) cpu-hours |
| Low-value job value | Gamma(12, 2.4) florins |
| High-value job value | Gamma(36, 7.2) florins |
| Low:high-value clients ratio | 80:20 |
| Delay before value decays | 1.5 $\times$ job-length |
| Mean decay rate (steep) | to 0 value in 1 job-length |
| Mean decay rate (shallow) | to 0 value in 5 job-lengths |
| Shallow:steep clients ratio | 80:20 |
| Max penalty value | $= -$(job-value) |
| Job-cancellation penalty | $=$ max-penalty-value |
| Resources available | 20 processors |
| Resource cost | 10 florins/hour |
| On- and off-period duration | 125 hours each |
| Quantity- and cost-ratio | 1.0 (static) |

## 5 Experimental setting

We studied our job-execution service provider's behavior across a wide range of operating conditions, varying the offered workload and contract conditions, the policies and algorithms used by the service provider, and resource-provider behavior. This section describes our experimental setup, and the default parameters used in our experiments. The next section presents the results we obtained.

We started by constructing a system with a set of independent Java processes to act as clients, a service provider and a resource provider. This system is able to perform real job-execution, for a set of "fake" jobs. That code base forms the basis of a simulator, which mimics the behavior of the real system, and was used for all the results reported here.

In our experiments we use a single job-execution service provider, renting computers from a single resource provider, as this is sufficient to stress our algorithms.

We begin with a set of exploratory runs that are designed to tease out the behavior of the system under relatively straightforward conditions, before proceeding to more challenging situations. This first set of experiments establishes

a baseline operating environment with the static resource provider: our goal is to set up a workload that has a reasonable profit margin (about 50%), operating at or near saturation on the available resources, while still rejecting some contracts and individual jobs. The first set of results we report present this behavior (see Section 6.1).

To allow comparison with prior work, we model our test workloads on the ones used by Millennium [7], RiskReward [14], and Popovici [21]. Each job utility function has a fixed value-decay rate (see Figure 2) that reduces the job's value from its initial value to its maximum penalty value; the decay starts at 1.5 times the job's running time. We use a mixture of high and low-value jobs and both shallow and steep value-decay rates. RiskReward[14] discusses how these synthetic loads relate to real workloads. Not all offered jobs can be executed profitably with the cost, computation time, and value settings used, even if the service provider is otherwise idle.

The default parameter-settings for our runs are shown in Table 1. Each client generates one sequence of jobs with an exponentially-distributed inter-arrival time. Such clients are created at exponentially-distributed inter-arrival times throughout the run, with a contract duration designed so that contracts may span on/off boundaries. Contract negotiation is simulated as occurring at the time a client is created; the client's first job is submitted one job inter-arrival time later.

We used Gamma distributions to generate bounded values such as job length or utility values.[7]

Unless otherwise noted, all graphs present averages over 10 runs. We take care to avoid end-effects as much as possible. In particular, we run all jobs to completion, make the experiment duration much larger than the average job duration, and undo the effects of jobs that are incomplete at the end of a run.

We found that the execution times for the algorithms of the job execution service provider are small compared to the time for running the jobs.

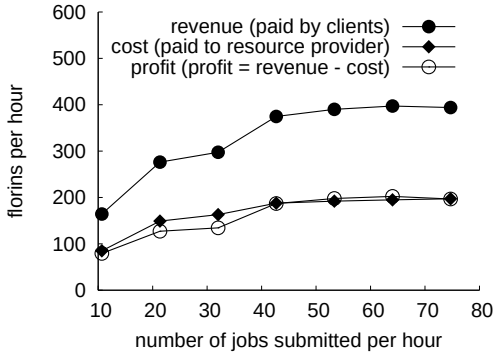
## 6 Results

We now present our simulation results. We begin by establishing the baseline behavior for our system in the absence of aggregate utility-aware clients, and then add them, followed by varying the behavior of the resource provider.

### 6.1 Baseline behavior

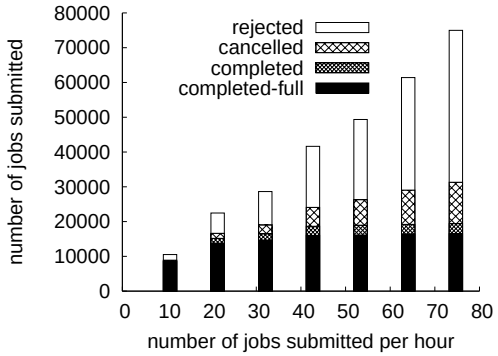
The policies used by the baseline service provider are *contract-load=oblivious*, and *contract-cost=any* for

[7] Gamma distributions are chosen with parameters such that they behave roughly like normal distributions but with the attractive property that they do not generate negative values. Using a normal distribution and suppressing such values would result in a new, not-quite-normal distribution with a slightly different mean than intended.




(a) revenue, profit and cost


(b) breakdown of cpu usage by job value-rate (florins/hour)




(c) breakdown of job fates; "completed-full" jobs are ones that earned their maximum value

**Figure 7. Indifferent clients and a static resource provider.** *Client utility rate (revenue), service-provider profit rate, utilization and breakdown of utilization versus offered load for the baseline service provider.*

contract-admission and *first-profit-rate* for job admission control and scheduling.

Figure 7 shows how the service provider and clients behave in the absence of an aggregate utility function (or, more strictly, if the aggregate utility function is "indifference"), and with a static resource provider.

As the offered load increases, the overall utility delivered to both clients and service providers increases. The service provider costs (which equal resource-provider revenues) stop growing significantly at around 60 jobs/hour, at the same point that resource utilization saturates.

Revenue and profit also stop growing at resource saturation. The service provider finds and runs more higher-valued (and hence more profitable) jobs at higher loads, as shown by the breakdown of cpus assigned to jobs of different values (Figure 7(b)). However, accepting higher-value jobs is achieved at the expense of cancelling more lower-value jobs and the net result is a flat profit curve.

Note that the service provider's tendency to cancel low-value jobs when higher-value jobs arrive is what motivated our desire for client aggregate-utility functions.

We used these first results to establish a baseline operating point for the remaining experiments. The parameters that resulted are shown in Table 1.

### 6.2 Aggregate utility functions

Figure 8 shows the effect of introducing client aggregate utility functions under different service-provider contract-admission policies (section 3.1). For this experiment, the aggregate utility function is $f(x) = x^2$ ($\alpha = 1$ and $\beta = 2$).

Figure 8(a) shows profit earned using each policy. In the presence of aggregate-utility aware ("sensitive") clients, profit drops dramatically as load increases for the oblivious admission control policy, which admits all contracts. The other three policies, which limit the number of contracts and hence the number of jobs submitted, see increasing profit with increasing load.

Figure 8(d) shows that while just as many (in fact, more) jobs are completed using the oblivious policy, there are also a lot of cancelled jobs. Fortunately, since first-profit-rate job scheduling always prioritizes the high-value jobs, virtually all (95-100%) of the jobs for high-value contracts are completed. At the same time, a much lower fraction of jobs are completed for low-value contracts. Combined with the penalty incurred for cancelled jobs, the lower fraction causes the overall revenue earned from the low-value contracts to result in negative profit. In fact, for low-value contracts, if only 90% of the jobs complete, the aggregate utility function reduces an average job value of 12 to 10 (which equals cost). When fewer than 90% complete, the *completed* jobs are actually run at a loss. The preempt-conservative policy, by contrast, finishes over 90% of the jobs for most of the low-value contracts that it accepts.

Figure 8(c) shows how many fewer contracts are accepted using the policies that monitor and limit load. The unrealistic high-value-only policy accepts the 20% of contracts that are high-value and completes nearly every job at full value, as seen in Figure 8(d). However, job arrival is bursty enough that even this policy cannot complete every high-value job, which is why there are small bands of completed (late) and cancelled jobs at the top of the bar. Furthermore, the much smaller number of jobs completed using the high-value-only policy as compared to the preempt-conservative policy shows that not enough contracts are accepted when using the high-value-only policy to keep the resources utilized.

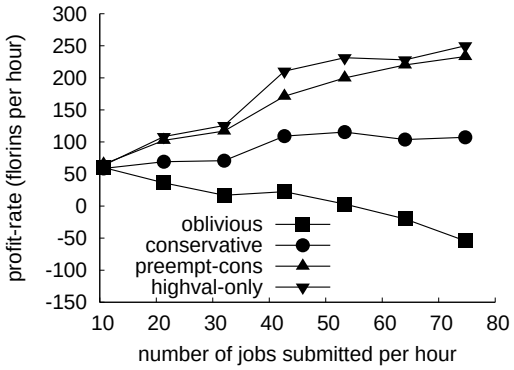
The two policies that consider load when performing contract admission, conservative and preempt-conservative, accept and complete about the same number of jobs. However, Figures 8(b) and (c) show that preempt-conservative is able to accept more contracts. By abandoning some contracts in favor of more profitable contracts that arrive later, the preempt-conservative algorithm sees a higher percentage of high-value jobs. These high-value jobs then have a large positive impact on its profit.

Note that profit for these experiments is lower for all policies than in the baseline experiments. By choosing an aggregate utility function where $\alpha = 1$ and $\beta = 2$, the effect of the aggregate utility function is always to diminish revenue. In other experiments where $\alpha = 2$, omitted here for lack of space, we see much higher (nearly double) profit.

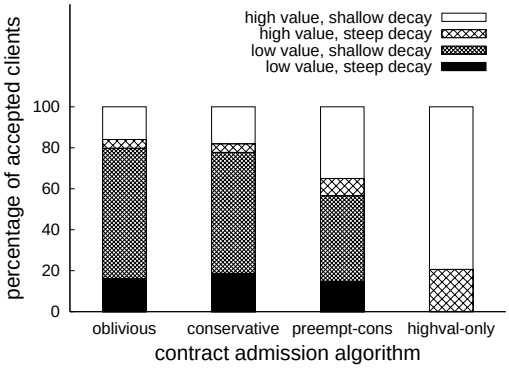
All the runs shown in Figure 8 use the effective job utility function described in Section 3.2 when constructing schedules, both for job admission and job scheduling decisions. The benefit of using the bias calculated by this function is shown in Figure 9, which compares the performance of the preempt-conservative policy with and without the bias. Adding the bias improves profit because about 5% more low-value jobs are accepted and run: while these jobs are individually unprofitable and hence rejected without the bias, completing these jobs raises the percentage of jobs completed and hence the aggregate utility bonus enough to more than compensate for their individual losses.

The results shown in Figure 8 for $\alpha = 1$ and $\beta = 2$ are similar with other values of $\beta > 1$, as shown in Figure 10. For larger values of $\alpha$ and $\beta$, adding the bias calculation has a larger impact on the service provider's profitability. This greater profitability is important for clients as well. Client satisfaction is measured in terms of value per accepted contract, and as we can see from Figures 8(a) and (c), the algorithms earning a higher profit-rate also deliver a greater level of client satisfaction.

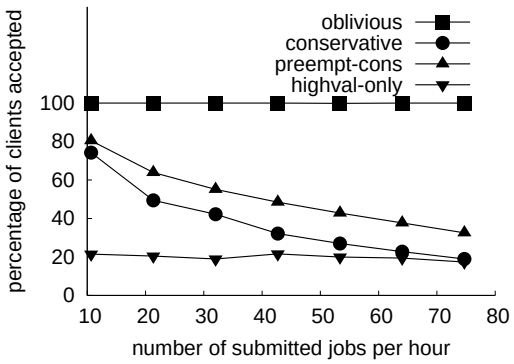
We use these results establish an operating point for the experiments relating to variability in the resource provider. For the remaining set of experiments, we use the *preempt-conservative* contract admission policy.



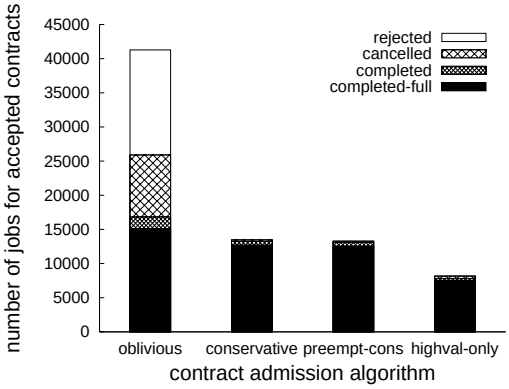

(a) profitability




(b) types of accepted clients, load=75 jobs/hour

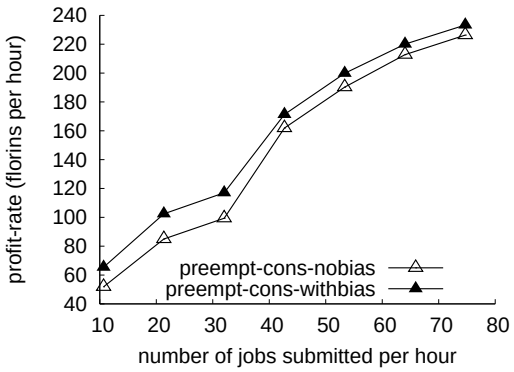



(c) number of contracts accepted

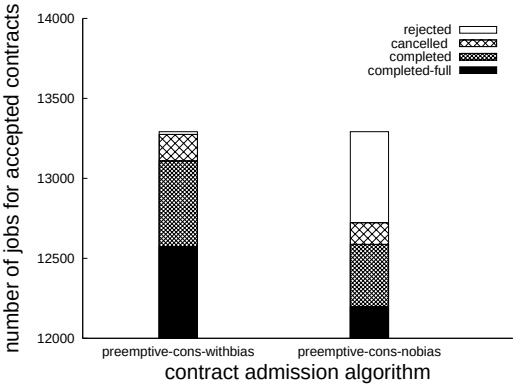



(d) job fates, load=75 jobs/hour

**Figure 8. Sensitive clients.** *Exploring the effects of different service provider contract-admission policies.*




(a) profitability




(b) job fates

**Figure 9. Job admission and scheduling.** *Enabling and disabling the effective job utility function.*

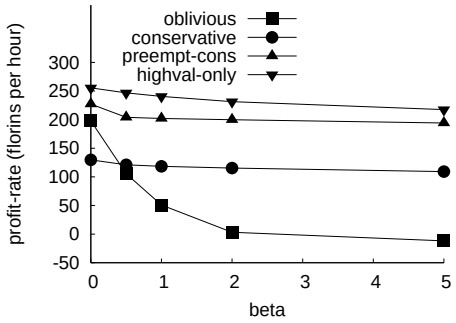



**Figure 10. Sensitivity of aggregate-aware clients.** *Exploring the effects of different choices of $\beta$ ($\alpha = 1$).*

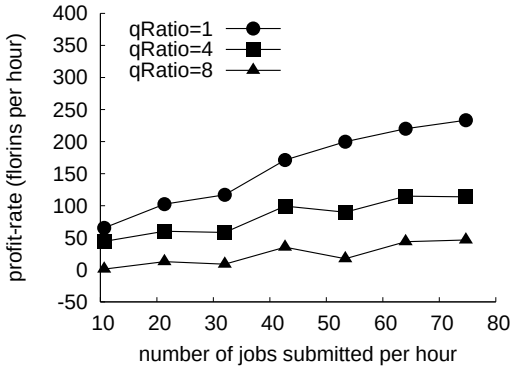
## 6.3 Static and variable resource providers

Figure 11 shows the effect of varying the resource provider behavior (Section 4) for the two contract-admission profitability tests described in section 3.1.

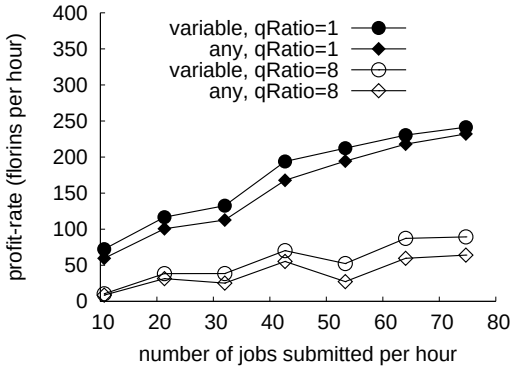
Figure 11(a) shows the results of fixing the resource costs, but varying the available resource quantity by changing the resource provider's quantity ratio (labeled as *qRatio* in the figures). A quantity ratio of 1 corresponds to the static resource provider used in the previous experiments.

As the quantity ratio increases, the profit rate of the service provider decreases. We hold the average number of resources constant, so when there are more resources in an on period, the number of resources in the off period drops accordingly. For a fixed level of demand, there is an abundance of profitable jobs during the low quantity periods and not enough jobs during the high quantity periods to use all of the resources. When the quantity ratio is 8, the contract completion rate is so low that most of the jobs from low-value contracts incur a loss, regardless of whether or not the job was actually run. This loss completely offsets the profit from the high-value contracts, which are also completing an average of only 85% of their jobs rather than the 98% they complete with static quantities of resources.

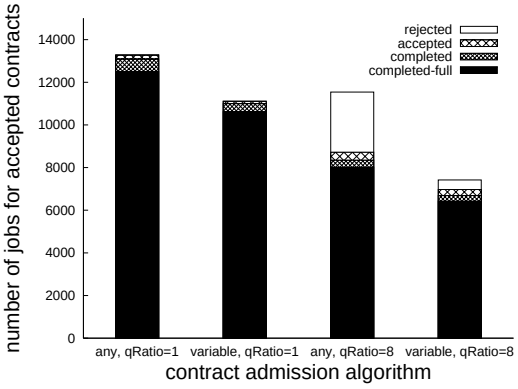
Figure 11(b) shows the effect of variable resource costs (by varying the cost-ratio) and also illustrates the benefit of turning on the profitability check from Section 3.1. As in Figure 11(a), increasing the quantity ratio decreases the profitability of the service provider. However, both with and without variable quantities of resources, the *contract-cost=variable* admission control policy outperformed the *contract-cost=any* policy, especially at higher offered load, since the provider is able to complete a larger fraction of jobs for a given contract. Note that a higher degree of client sensitivity (i.e. larger $\alpha$ or $\beta$) would increase the profit gap between the two policies.




(a) Cost-ratio = 1; the results are not affected by the choice of profitability test.




(b) Cost-ratio = 1/8; profit rate for two different profitability tests.




(c) Cost-ratio = 1/8; job fates for two different profitability tests, load=75 jobs/hour

**Figure 11. Aggregate-aware clients and a variable resource provider.** *The effects of different quantity-ratios and contract-admission profitability algorithms; $\alpha = 1, \beta = 2$.*

### 6.4 Summary

We used the initial explorations to establish a workload mix that suitably stressed the job execution service running on a static resource provider.

Adding in aggregate utility functions demonstrated the importance of being aware of client goals. Policies that took such client needs into account did significantly better than ones that did not.

We then explored how the variable resource provider's behavior affected the system's behavior, and saw that taking account of changing resource availability and cost gave much better results than being oblivious to such concerns.

Out of all our designs, we recommend the use of a *preempt-conservative* contract admission algorithm with a *variable cost* profitability prediction, and the *effective-utility bias* job scheduling approach. Together, these represent a good balance between effectiveness, robustness, and ease of implementation and execution.

## 7 Related work

Our work extends three previous studies in this area: Millennium [7], RiskReward [14], and [21]. Both Millennium and RiskReward merged the role of the service provider and resource provider and assumed a fixed-sized pool of nodes on which to run jobs. In doing so, they did not explicitly consider resource costs or resource variability. [21] used a system model similar to ours with a service provider that rents resources instead of owning them. We directly build upon this work, but instead of exploring the effects of resource availability uncertainty, we explore the effects of known variability. Unlike our work, none of these three studies considered aggregate performance constraints.

Our work is part of the larger field of computational economics, or agorics, which uses market models to determine the pricing of services and resources (e.g., Spawn [23], Mariposa [22], and Tycoon [17]). Our work extends these ideas by applying the notion of charging to other service types as well as resource rentals, but does not explicitly consider price-setting mechanisms such as auctions.

As in much prior work, we use per-job utility functions to specify the client's value of job completion. This is a well-trodden field: the Alpha OS [8] handled time-varying utility functions; [6] discussed how to do processor scheduling for them; [18] looked at tradeoffs between multiple applications with multiple utility-dimensions; Muse [5] and Unity [24] used utility functions for continuous fine grained service quality control; and [20] described using utility functions to allow speculative execution of tasks. Unlike prior systems, our clients use aggregate utility functions to control service provider behavior across multiple jobs. [16] used a single utility function that aggregated data from multiple sources. We use both per-job and aggregate utility functions.

Our client contracts indicate how a service provider should perform in the face of changing underlying conditions and conflicting service contracts. Off-line bilateral contracts [1] have been used to specify service quality for distributed stream-processing applications; SNAP [9] performs service and resource allocations based on three levels of agreements for resource management in grid computing: application performance, resource guarantees, and binding of applications to resources. Unlike SNAP, our job-execution service, rather than the client, determines how to bind applications to resources. [10] is similar, but also includes client level objectives and abstract resource objectives, and focuses on the service provider's need to manage resources and applications; however, they do not look at the effects of aggregate objectives nor the performance of a service provider to meet their objectives. GRUBER [11] uses contracts for job scheduling based on the amount of CPU a group is allowed to consume over a period of time, but does not assign values to individual jobs.

## 8 Conclusions

We investigated a portion of the design space of service providers and clients that wish to control aggregate-level behavior in addition to per-work-item actions. We did so in the context of an upper-level service provider that rents resources rather than owns them, and has to handle varying resource availability and cost.

We explored the effects of profit-aware algorithms, and studied how load, aggregate utility functions, and the number and cost of resources influenced the service provider's profit.

We showed that our profit-aware approach outperforms previous ones across a wide range of conditions, and make the following additional observations:

- The idea of a self-interested, profit-aware service provider is a powerful technique for thinking about, generating and selecting algorithms, and avoiding imprecision in defining a "good" outcome.
- The contract-admission control algorithms we developed seem to be quite effective, and our evaluation highlighted how important careful selection of work is for a service provider.
- Successfully handling aggregate utility functions is a new result. Doing so also increases client utility, which is an important result in its own right.

Our results demonstrate the importance of profit-aware schedulers and admission-control algorithms, and include cases where a decent profit could be obtained in place of losses from less profit-sensitive algorithms.

## References


[1] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *Proceedings of 1st Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[2] A. Baldwin, Y. Beres, D. Plaquin, and S. Shiu. Trust record: high-level assurance and compliance. In *Proceedings of iTrust 2005*, volume 3477 of *Lecture Notes in Computer Science*, pages 393–6. Springer Verlag, May 2005.

[3] J. Beckett. Hp labs goes hollywood: researchers help bring "shrek 2" to life, April 2004. Web page.

[4] A. C. Benjamim, J. Sauvé, W. Cirne, and M. Carelli. Independently auditing service level agreements in the grid. In *Proceedings of 11th HP OpenView University Association Workshop (HPOVUA 2004)*, June 2004.

[5] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centres. In *18th ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[6] K. Chen and P. Muhlethaler. A scheduling algorithm for tasks described by time value function. *Real-time Systems*, 10(3):293–312, May 1996.

[7] B. N. Chun and D. E. C. (UCB.). User-centric performance analysis of market-based cluster batch schedulers. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2002.

[8] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the alpha real-time distributed kernel. In *Winter USENIX Technical Conference*, April 1993.

[9] K. Czajkowski, I. Foster, C. Kesselman, V. Sander, and S. Tuecke. SNAP: a protocol for negotiating service level agreements and coordinating resource management in distributed systems. *Lecture Notes in Computer Science*, 2537:153–83, January 2002.

[10] A. Dan, C. Dumitrescu, and M. Ripeanu. Connecting client objectives with resource capabilities: an essential component for grid service managent infrastructures. In *Proceedings of 2nd International Conference on Service Oriented Computing (ICSOC'04)*, November 2004.

[11] C. L. Dumitrescu and I. Foster. Gruber: a grid resource usage sla broker. *Lecture Notes in Computer Science*, 3648:465–74, August 2005.

[12] A. Elfatatry and P. Layzell. A negotiation description language. *Software—Practice and Experience*, 35(4):323–43, April 2005.

[13] I. Foster and C. Kesselman, editors. *The Grid 2: blueprint for a new computing infrastructure*. Morgan Kaufmann, 2nd edition, 2003.

[14] D. E. Irwin, L. E. Grit, and J. S. Chase. Balancing risk and reward in a market-based task service. In *Proceedings of 13th IEEE Symposium on High Performance Distributed Computing (HPDC)*, June 2004.

[15] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *Proc Conference on File and Storage Technologies (FAST'04)*, March–April 2004.

[16] V. Kumar, B. F. Cooper, and K. Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *Proceedings of 2nd International Conference on Autonomic Computing (ICAC'05)*, June 2005.

[17] K. Lai, L. Rasmusson, E. Adar, S. Sorkin, L. Zhang, and B. A. Huberman. Tycoon: a distributed market-based resource allocation system. Technical report, Hewlett-Packard Laboratories, Palo Alto, CA, September 2004.

[18] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A scalable solution to the multi-resource qos problem. In *20th IEEE Real-Time Systems Symposium (RTSS'99)*, December 1999.

[19] P. Patrick. Impact of soa on enterprise information architectures. In *Proceedings of 2005 ACM SIGMOD International Conference on the Management of Data*, June 2005.

[20] D. Petrou, G. R. Ganger, and G. A. Gibson. Cluster scheduling for explicitly speculative tasks. In *Proceedings of International Conference on Supercomputing (ICS'04)*, June–July 2004.

[21] F. I. Popovici and J. Wilkes. Profi table services in an uncertain world. In *Proceedings of Supercomputing (SC|05)*, November 2005.

[22] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. An economic paradigm for query processing and data migration in mariposa. In *International Conference on Parallel and Distributed Information Systems (PDIS)*, September 1994.

[23] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: a distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–17, February 1992.

[24] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *Proceedings of 1st International Conference on Autonomic Computing (ICAC'04)*, May 2004.