Precise and Realistic Utility Functions for User-Centric Performance Analysis of Schedulers

Cynthia Bailey Lee Allan E. Snavely
Dept. of Computer Science and Engineering and San Diego Supercomputer Center
University of California, San Diego
La Jolla, CA 92093-0505
cl@sdsc.edu allans@sdsc.edu

ABSTRACT
Utility functions can be used to represent the value users attach to job completion as a function of turnaround time. Most previous scheduling research used simple synthetic representations of utility, with the simplicity being due to the fact that real user preferences are difficult to obtain, and perhaps concern that arbitrarily complex utility functions could in turn make the scheduling problem intractable. In this work, we advocate a flexible representation of utility functions that can indeed be arbitrarily complex. We show that a genetic algorithm heuristic can improve global utility by analyzing these functions, and does so tractably. Since our previous work showed that users indeed have and can articulate complicated utility functions, the result here is relevant. We then provide a means to augment existing workload traces with realistic utility functions for the purpose of enabling realistic scheduling simulations.

Categories and Subject Descriptors
D.4.9 [**Operating Systems**]: Systems Programs and Utilities –*scheduling of parallel supercomputers, model of users' valuation of their jobs, resource management.*

General Terms
Algorithms, Management, Measurement, Performance, Economics, Human Factors.

Keywords
Utility Functions, Scheduling, Genetic Algorithms.

1 INTRODUCTION
We claim that high-performance computing (HPC) and grid batch schedulers exist primarily to maximize user satisfaction, and that system owners' needs are best met when users are most satisfied. There are many aspects to satisfaction, including speed of job turnaround, predictability of job turnaround times, interface ease-of-use and even aesthetics, and others. This work focuses on the value users associate with their jobs' turnaround time, specifically value (also called utility) as a function of time elapsed from when the job was submitted. This function, *u(t)*, is referred to as a utility function, and is a concept widely used in the field of economics which has recently been applied in HPC and Grid scheduling research.

A reasonable scheduler should be most effective at maximizing user utility when it has access to the most information about what users want. However, previous research on scheduling and user utility has generally restricted the shape of *u(t)* to a simple linear format. In previous work, we demonstrated that users had complex utility functions that could not be captured well using the simple linear format. This paper advocates the use of a more detailed and informative utility function, including providing evidence that the scheduling problem using these functions can indeed be solved (heuristically) in timeframes that allow practical usability.

2 PREVIOUS WORK
Opportunities for HPC users to communicate information about their scheduling preferences to schedulers are limited. On most production HPC systems, they are merely able to indicate a priority for their jobs. Priorities are typically selected from a short list of discrete options, e.g. *Low, Normal, High*.

Ironically, users' freedom to express priority has decreased over the years. More fine-grained priority choices were commonplace on SMP supercomputers of the 1980's and 90's. For example, jobs on the Cray XMP were assigned a floating-point priority value between 0 and 2 by their owners. Users could change a job's priority at any time and as often as they pleased, whether the job was being held or running. [16]

Although users of production systems have limited opportunity to express priority and preferences, proposed schedulers in the research literature have taken steps to allow more flexibility. Many of these generalize and unify the communication of preferences under an economic scheme. Stoica, Abdel-Wahab and Pothen [18] proposed a Microeconomic Scheduler that allows users to create expense accounts for their jobs, thereby expressing the job's priority. The system, in turn, can implement incentives for

desirable behaviors from users, such as charging users not just for the time a job uses, but for the idle time created in the job's wake, thus enlisting users in the effort to decrease fragmentation in the schedule. Feitelson and Rudolph [8] have postulated that the diverse sub-research-areas of batch job scheduling (e.g. gang scheduling, dynamic partitioning) can converge under a framework with a flexible economic-based philosophy such as this, and the work of Wolski et al. [24] also points to an economic model. There are a number of projects in the Grid realm that are following this trend, such as Mirage [3] and Tycoon [11] and Spawn [22].

Some more recent proposed parallel batch job schedulers and scheduler evaluation schemes [10, 2, 4, 1] have employed simple linear utility functions, with customizable maximum (starting) value and slope (as shown in Figure 1 below). The form used in [10] also allows for a penalty for "late" completion of jobs, with the decay in value continuing on the same trajectory.

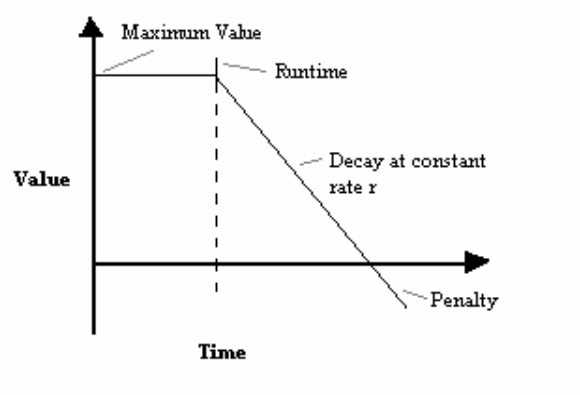


Figure 1. Job utility function as used in [10].

In this formulation, the utility function is essentially not defined during the time the job is running—the loss of value that occurs during this time is not represented. This is adequate when narrowly considering scheduling on a single fixed system with consistent runtimes (the passage of time due to running the job is unchangeable and unavoidable).

However, it is obvious that users do value faster runtime, not just shorter wait. If this were not so, nobody would purchase new, faster computers. Representing loss in value due to runtime would allow quantifying the value to users of such purchases. Upgrades of HPC systems can cost millions of dollars, so calculating this utility impact for users may be quite important. Another application would be quantifying the tradeoff in user utility of scheduling jobs to share processors or other resources, thus decreasing wait time but increasing runtime (such a scheduler has been proposed in [23]).

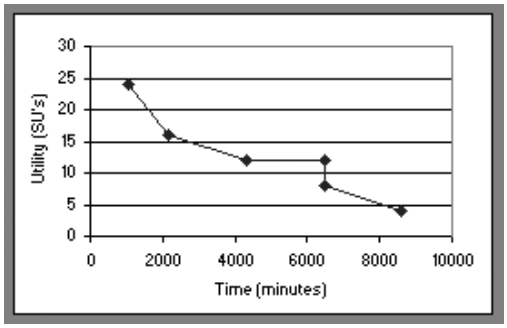
3 A NEW UTILITY FUNCTION REPRESENTATION

3.1 Is it needed?

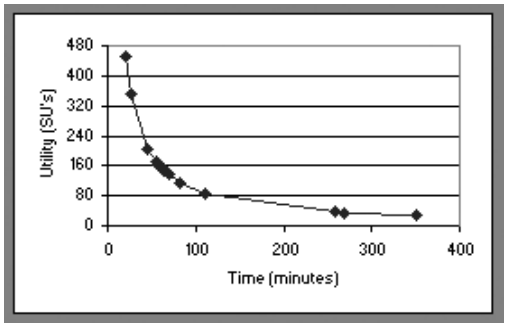
Might a new and more complex utility function representation be needed? Feitelson et al. [9] present an example scenario to use as a starting point for thinking about the desires of users:

> Assume that a job i needs approximately 3 hours of computation time. If the user submits the job in the morning (9 a.m.) he may expect to receive the results after lunch. It probably does not matter to him whether the job is started immediately or delayed for an hour as long as it is done by 1 p.m. Any delay beyond 1 p.m. may cause annoyance and thus reduce user satisfaction…. However, if the job is not completed before 5pm it may be sufficient if the user gets his results early next morning.

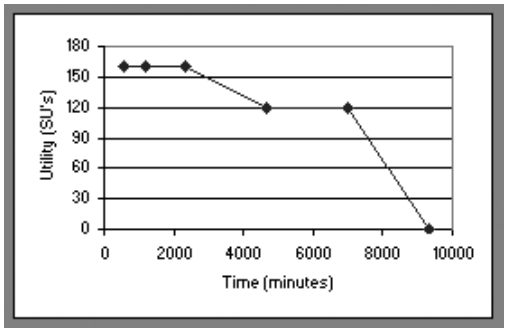
The utility function *u(t)*, implied by this scenario is not a simple linear function, like that of [10]. There are discontinuities (e.g. at 1 p.m.) and periods where the slope is zero (e.g. between 5 p.m. and the following morning). Furthermore, it is likely that every *(user, job)* pair will have a function with a different pattern.



(a)



(b)



(c)

Figure 2(a-c). Utility functions collected from users. (from [12])

In previous work, we have conducted a survey of users' valuations for real jobs on a real HPC system. The results are detailed in a previous paper [12]. Briefly, users of a system at the San Diego Supercomputer Center (SDSC) were asked by the job submission program how many charge units (SUs) they would be willing to spend for faster or slower service. These data points were used to construct a piecewise-linear utility function for each user's job.

Figure 2(a-c) shows a few examples of the utility functions we collected. They are indeed more complicated than the simple linear formulation can capture, showing zero-slope regions and other patterns, as predicted in the example scenario above.

There is a legitimate concern that users may not be willing to provide this level of detail in a real-life job submission setting. They already struggle to provide the information currently asked of them, such as job runtime estimates [13]. The colloquial version of our opinion on this matter is that people may not be good at talking about their jobs, but if there is one topic everyone loves talking about, it is themselves. A utility function is not a property of the job; it is a property of the user. More concretely, our previous work shows that such information is obtainable, and our future work involves detailed human factors studies to fine tune the details of an agreeable interface for providing it in a real job submission setting.

3.2 Proposed New Formulation

We therefore propose a continuous piecewise linear utility function representation, that allows users to specify the location of each data point, and even the number of points to provide. This format is both rich in descriptive power, and simple to understand. Some of the possibilities for user self-expression afforded by this format are demonstrated in Figure 3.

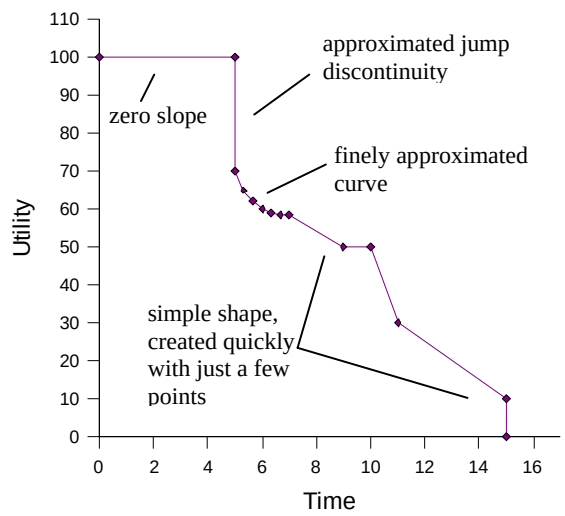


Figure 3. Demonstration of flexibility of proposed formulation.

Notice, in Figure 3, some of the features this formulation allows to be expressed: periods of zero slope ($0 \le t \le 5$ and $9 \le t \le 10$), approximated jump discontinuities ($5.0 \le t \le 5.001$ and $15.0 \le t \le 15.001$), a finely approximated curve ($5 \le t \le 7$) and a pattern perhaps more crudely drawn ($7 \le t \le 15$). The placement of points is user-determined both in terms of the x-dimension and the y-dimension. The number of points to include is also user-determined (minimum two).

3.3 Why This Formulation?

A main goal of this work was to allow a high level of detail in specification of the utility function. This format achieves that, in particular it is much more powerful than the purely linear (height and slope adjustable) formulation used in some previous work. On the other hand, we are mindful that, in order for the scheduler to obtain a utility function for a job, we are placing a time and energy burden on users, who must stop to reflect on their desires and then enter the information.

Although many HPC users are trained in the sciences and would be familiar with various standard function shapes (linear, exponential, hyperbolic, etc.), we believe it unwise to rely on this knowledge by requiring users to input formula parameters for these shapes. First, submitting a job should not require a mathematics glossary or formula reference. Second, we do not wish to impose a predetermined function shape on the user. Thus it meets the two essential requirements: it is easy for any user to input and flexible.

Piecewise linear also has an intuitive mapping to the type of typical daily routine-linked changes in value we saw in Feitelson et al.'s example scenario and in our survey of real users.

Finally, this formulation has the benefit that users can put forth as much or as little effort as they please. From a single line segment of two points, to a finely approximated curve composed of many segments, users can tailor their level of effort to the benefit they perceive can be extracted from the scheduler.

3.4 Detailed Format Specification

We store the function as a series of *(time, value)* tuples. The *time* in the first tuple must always be set to zero, and this represents the time at which the job was submitted. The *value* in the first tuple is the initial (maximum) value for the job if it were completed instantaneously. The units of value depend upon the system. Allocation units or service units ("SUs") are already in use at many HPC centers and grid sites, but in commercial and other settings, it may be units of real currency (e.g. US Dollars). The domain for both times and values is the set of non-negative real numbers. Reading the series from start to finish, times must be strictly increasing, while values are decreasing (non-strictly, i.e. periods where the slope is zero are allowed). Jump discontinuities are approximated by using two different times t and $t + \varepsilon$, with some very small ε.

We define the value at any time greater than the last time listed in the sequence to be zero (the sequence may also explicitly contain time(s) at which the value is zero).

Using simple linear interpolation between the user-provided data points, the sequences of tuples are interpreted as continuous piecewise-linear functions.

Thus all well-formed function specifications represent functions that are defined over the domain of all times t, $0 \le t \le \infty$. (Note that since $t = 0$ is the submit time of the job in question, times must be normalized to some absolute clock for comparison amongst different jobs.)

4 SYNTHETICALLY GENERATING UTILITY FUNCTIONS FOR WORKLOADS

The following is a proposed model of generating synthetic utility functions that, to the extent possible, incorporates knowledge and research about real user preferences. A software implementation of the utility function generation methods described here is available; please direct inquiries to the authors.

We believe strongly in using real workload data as much as possible. Tsafrir [20, 21] and others [19] have elucidated the many subtle attributes of real workloads and how they can have unexpectedly significant impact on performance. We look forward to a time when workload traces including users' own utility functions are available, but we find it necessary to propose this statistical model in the interim, in order to perform the simulations necessary to justify asking them of an entire user population.

4.1 Input Data

Workloads in the *Standard Workload Format (SWF)* form the basis of our synthetic workload generation. Many logs of actual HPC workloads are available in this format from the Parallel Workloads Archive [7, 5]. Each line of the file represents one job. Standard job description data such as number of processors, submit time, runtime, priority, and so on, are provided. We extend the *Standard Workload Format* by appending the list of tuples that define our utility function to the end of each line.

We have three sources of information to draw on when forming a model for generating utility functions: 1) each job's actual user-assigned priority and completion time, 2) what we learned about utility function shapes from our previous study [12], and 3) the distribution of actual wait times from the log, which gives guidance about user expectations.

The priority we find in the workload data is the main clue we have as to how the user values that specific job. However, this is at best a loose guide to the initial value of the utility function and none at all to its overall shape. Our vision is that the user has the equivalent of a utility function in mind simply by virtue of being a human with needs and desires. In selecting a priority for the job, the user was forced to project that utility function curve onto a single dimension, one with a severely limited domain at that (e.g. 3 or 4 choices). It is not clear exactly how this projection of the function was or should be carried out. If a user's job has a high maximum value, indicating importance, but retains its value for a long duration, indicating lack of urgency, would that user have assigned it "Low" or "High" priority? There is no obvious answer, and probably different users have different methods of doing this mapping. Some may effectively have no method at all—they may not see an appropriate choice for expressing their needs and wants, and just arbitrarily select one of the options. Details of how we use this data are in the next section, but let us emphasize that we do not claim to perform the reverse of this projection in a way that is historically accurate in each individual case. This would be impossible. We only seek to generate utility functions that do not *contradict* the available data.

We also incorporate what we learned about utility function shapes from the previous study already mentioned. Again, this information is a loose guide. Each system, user and job are unique, and we only have data for a certain group at one site. But our synthetic utility function model attempts to incorporate commonly seen traits from the collected curves.

Each workload has a different priority scheme: number of different priorities, restrictions on eligibility of jobs for different priorities (commonly, a limit on number of processors or time), etc. These reflect policy choices by the administrators of the various systems. In the SWF format, priority is encoded as an integer, with header comments giving meaning to the values. Some human interpretation of this will be needed for each different workload.

Our model currently incorporates a generalized version of commonly seen priority systems where the different priorities have a strict lowest-to-highest ordering that makes sense according to the semantics given in the header comments of the SWF file. We also assume that the "step" between each increasing priority is of equal distance in terms of value to the user. We normalize the representation of the priority choices to be natural numbers 0, 1, …, *Npriorities-1,* where 0 signifies the highest priority and *Npriorities-1* the lowest (*Npriorities* is the number of different priorities in the system). For example, *Low, Medium, High* would be normalized to *Low*=2, *Medium*=1, *High*=0.

4.2 Generating the Starting (Maximum) Value for Jobs

First, we determine what the maximum (starting) value for the job will be. This is based on the priority for the job and its size (number of processors times requested time).

The space between zero and *globmax* (an arbitrary global maximum value) is divided into *Npriorities* bins of equal size, where *Npriorities* is the number of priority choices available on the system in the input workload. We want to reflect that, in general, higher priority jobs should have higher start values than lower priority ones. So we randomly select the starting value, $startvalue_{unscaled}$, for the job using a Normal distribution, centered over the bin corresponding to the job's priority (as shown below in the formula for *mean*). This causes most, but not all, jobs to have a start value proportional to their priority.

$$mean = ((priority + 0.5) / Npriorities) * globmax$$

This randomly selected starting value for the job is the value per processor-minute requested. So to compute the value for the entire job, we factor in the job's size:

$$startvalue = startvalue_{unscaled} * processors * time$$

4.3 Modeling Decay Patterns

The first tuple in our utility function is *(0, startvalue),* as described above. To generate the remaining tuples, we use three different models of decay in value: *expected linear, expected exponential,* and *step.* These represent roughly the three cateogories of patterns we observed in our survey of actual user utility functions.

Expected linear represents users whose value decays mostly linearly over time. *Expected exponential* represents users whose jobs have high initial urgency (a much more precipitous initial decline); however, after some time has passed, additional delay causes a diminishing marginal decay in utility. In both cases, *expected* refers to the possibility of some "bounce" or random out-of-pattern behavior. *Step* represents users whose jobs remain at their peak value for some amount of time, then experience a step-function-like immediate drop in value, then after another flat period, lose all their value.

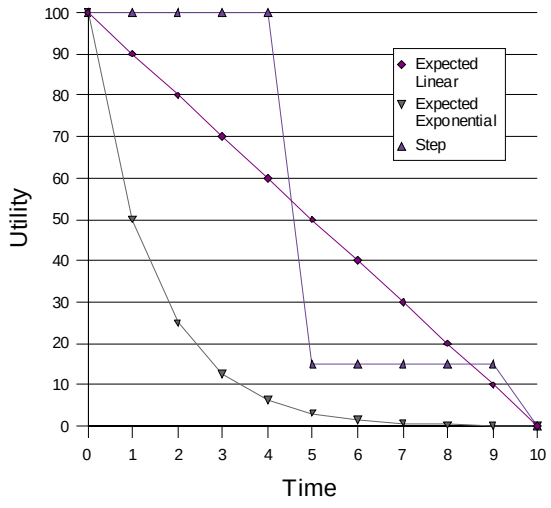


Figure 4. The statistically *expected* pattern of utility functions generated using three different decay types. Compare to Figure 2(a-c).

To generate these decay patterns, we first select the time at which the job will finally lose all of its value, called the *deadline*. We want *deadline* to reflect, as closely as possible, real user expectations for wait times for their jobs. This is dependent on the priority chosen, but also heavily dependent the job's size (number of processors and duration). To capture the nature of this dependency and real expectations of distribution of wait times, we base the deadline on the actual time, from the log, that the job originally waited, setting *deadline* to be twice this time (or 10 seconds, whichever is greater).

Now we have the first and last times and values of the sequence, and we select the intermediate times and values. For expected linear and expected exponential, we randomly select a predetermined number of unique times between zero and *deadline.* For step, we randomly select one intermediate deadline between zero and *deadline.* The times are then sorted in increasing order. The values corresponding to this sequence of times are selected as follows. For expected linear, we select values randomly between zero and *startvalue,* inclusive. For expected exponential, we iteratively select values randomly between the last value selected (or *startvalue)* and zero, inclusive. Values do not need to be unique. Values are sorted in decreasing order and matched to the sequence of times to complete the tuples.

5 THE AGGREGATE UTILITY METRIC

Classic scheduling metrics such as *average wait time, expansion factor,* or *bounded slowdown,* and, to a lesser extent, *makespan*, are all intended to represent the notion that good schedulers are those that maximize user happiness, i.e. minimize users' frustration due to being made to wait for results. *Average wait time, expansion factor,* and *bounded slowdown* all implicitly assume a plain linear decrease in value to users over time, with all users and jobs having the same start value and rate of decrease.

As noted in, e.g. [10, 2, 4, 1], when armed with workloads bearing utility functions, we can directly compute the total value delivered to users:

$$Aggregate\ utility = \sum_{j \in Jobs} utility_j(turnaround\_time_j)$$

This metric necessarily gives more weight to fast turnaround of some jobs over others. However, this cannot be considered an unfair preference, as we assume that this weight is the correct value, according to the priorities of both the system administrators and the individual users. For this to be true, the ability of users to specify high value in their utility functions must be moderated by externally imposed limits that reflect administrator goals (e.g. finite allocation), and users must have been truthful in their specification of their utility functions.

We will compare various well-known scheduling algorithms by their ability to maximize aggregate utility. First, for purposes of comparison, we introduce a scheduler that explicitly uses aggregate utility as an objective function.

6 AGGREGATE UTILITY AS AN EXPLICIT OBJECTIVE FUNCTION FOR SCHEDULING

For purposes of comparison to classic scheduling algorithms, we introduce a simple scheduler that uses a genetic algorithm (GA) heuristic to evolve a queue ordering of the jobs seeking to maximize aggregate utility. Partly, we introduce this algorithm simply as a way to explore the question, "How well is it possible to do?" In other words, what is the opportunity-space to maximize aggregate utility under realistic assumptions about users' utility functions? Although the notion of using the GA scheduler on a real system is not implausible, we emphasize the prototype-level condition of this particular implementation. Use of it on a real system would require additional refinement and is an element of our future work.

Our measure of fitness for a given queue ordering is an estimate of the aggregate utility that would result if the jobs

were scheduled in that order according to the EASY backfilling algorithm.

Reproduction is done by combining the two parent queue orderings in a way analogous to a zipper. The first job in one parent's queue is popped and becomes the first job in the child queue. Then the second job of the child queue is the first job from the other parent, and so on. If the next job to be added to the child's queue is already present in the child's queue, the job is popped from the parent queue and the search continues until a non-duplicating job is found.

Point mutations are simulated by selecting two jobs in the queue at random, and swapping their places.

The population is seeded with random permutations of the job queue, and a few special seed individuals whose jobs are sorted by, variously: arrival time, priority (secondarily by arrival time), current utility per node-hour, and absolute current utility.

The GA scheduler was written in python, adapting python code [15] provided in conjunction with the textbook by Russell and Norvig [17]. All scheduling algorithms in this paper were implemented within a common framework, written in python.

One drawback to using an optimization approach like genetic algorithms, is that it is not deterministic. Fortunately, performance of GA was highly consistent across runs. For example, on 10 repeated trials using the exact same workload, aggregate utility ranged from 1.32×10^9 to 1.38×10^9, with a standard deviation of 1.72×10^7, or 1.27%. Because variation is possible, results for GA presented hereinafter are the average of two runs of GA per synthetic utility function augmentation of the workload in question (also done twice), for a total of 4 trials per test or configuration.

Of course, the major drawback to using a heuristic optimization instead of an efficient deterministic algorithm is the much longer execution time. We bound the number of generations of reproduction at a relatively modest 100 and use just 20 individuals in the population. Still, on a workload of 5,000 jobs, GA took an average of 8,900 seconds, compared to just 110 seconds for conservative backfilling, 35 seconds for Prio-FIFO and 30 seconds for EASY. Code design was not done with an emphasis on performance, and timings were performed on [GET PROC STATS] PC, but these numbers give some idea of the relative performance.

Although slower than other algorithms, performance of our GA scheduler is adequate to the task of real-time scheduling of a contemporary HPC system. In that environment, the scheduler must select which job(s) to start whenever a new job arrives or a running job terminates. The current version of GA takes 2 seconds to do this calculation. According to SDSC system logs, about 90 seconds elapses between jobs as the system “cleans up” and does other tasks, so this is well within reason.

7 COMPARING SCHEDULERS WITH AGGREGATE UTILITY

The following experiments were conducted using a workload with utility functions generated using the synthetic utility function generation methods described above. The base workload is SDSC-Blue [6], available from the Parallel Workloads Archive [7, 5]. SDSC Blue has 1,152 processors and six main priority categories. We use jobs 5,000 through 10,000, thus avoiding the atypical patterns at the beginning of the trace when the machine was first being opened to users. This represents just a few hours short of three full weeks of operation.

The schedulers to be compared are several of the classic scheduling algorithms: EASY backfilling, Conservative backfilling, and Priority-FIFO (simple supercomputer scheduler policy where all jobs from a higher priority queue are considered before any in a lower priority queue and EASY backfilling within these constraints). These are compared against the utility-function-aware GA scheduler.

Generation of synthetic utility functions, as described above, relies in part on pseudorandom selection of parameters; even for the same base input workload, resulting utility functions will differ each time they are generated. Thus all results reported in this paper are the average of the result on two different workloads (same base workload, augmented with utility functions two different times). Recall that GA has its own variability, and is run four times, twice on each of these two workloads.

7.1 Comparing Aggregate Utility to Other Metrics

The following chart compares some classic job scheduling algorithms, using common metrics, and also the aggregate utility metric, using a workload augmented with synthetically generated utility functions (default configuration).

Quickly, a description of the metrics used here:

- Wait Time – number of seconds a job waited in the queue.
- Expansion Factor – ratio of turnaround time (wait time plus runtime) to runtime.
- Average Bounded Slowdown – like Expansion Factor, but bounds the runtime at 10 seconds minimum, to limit the impact of extremely short runtime jobs on the average. Calculated by method in [14], which ignores first N divided by 101 and last N modulo 100 jobs in the trace (where N is the total number of jobs).
- Utilization – average over the duration of the simulation of the fraction of processors currently running a job.
- Makespan – number of seconds between the start of the simulation and when the last job completes.
- Percent of Job’s Start Value Earned – ratio (percent) of the job’s earned utility, (or $u(t)$ where t is the complete time of the job) and the maximum (starting) value for the job.
- Aggregate Utility – sum, over all jobs, of the earned utility of the job.

Examining the impact of the scheduler on the full population of jobs, not just an average, gives much richer insight. Therefore we present the results for several of the metrics as percentile ranges (e.g. 75% of jobs had a Wait Time of 73,881s or less using the Conservative backfilling algorithm).

Table 1. Comparison of Different Schedulers by Different Metrics

	Wait Time (s)* 25 %ile 50 %ile 75 %ile 98 %ile 100 %ile	**Expansion Factor*†** 25 %ile 50 %ile 75 %ile 98 %ile 100 %ile	**Average Bounded Slowdown***	**Utilization****	**Makespan (s)***	**% of Job's Start Value Earned**** 25 %ile 50 %ile 75 %ile 98 %ile 100 %ile	**Aggregate Utility****
Cons	0 1,375 16,746 73,881 112,103	1.0 1.30 2.25 34 146	131	0.360	3.742E+6	78.9 % 11.5 % 0 % 0 % 0 %	8.77E+08
EASY	0 0 5,372 59,067 116,552	1.0 1.0 1.55 28 131	76	0.360	3.736E+6	90.4 % 40.8 % 0.055 % 0 % 0 %	1.16E+09
Prio	0 0 2,733 72,403 319,822	1.0 1.0 1.30 22 353	73	0.361	3.729E+6	93.3 % 53.1 % 1.04 % 0 % 0 %	1.27E+09
GA	0 0 822 62,758 1,355,885	1.0 1.0 1.12 34 2261	148	0.365	3.690E+6	96.0 % 59.1 % 2.84 % 0 % 0 %	1.35E+09

* Smaller values indicate better performance on these metrics.
** Larger values indicate better performance on these metrics.
† 1.0 is the best possible value for this metric.

Of the classic algorithms, Priority-FIFO performed the best on all workloads, and on all metrics except expansion factor. Priority-FIFO and GA have made an explicit choice to favor some jobs over others, using the priority information associated with the job. *This is an explicit choice and we do not in any way consider this "unfair."* After all, the users themselves had a part in determining the utility functions for their jobs, which resulted in the differential treatment. (Only in the allocation of funds in the first place can the system be unfair; this is outside the scope of our examination.) We particularly note that Priority-FIFO and GA have not negatively impacted utilization or makespan (the contrary in fact); the hardware is being utilized just as efficiently by these algorithms, they are just making different choices as to which jobs to run first, guided by users' valuations.

Priority-FIFO and GA significantly outperform Conservative and EASY for aggregate utility. This is not surprising since they are the only algorithms that can take priority or utility into account. GA outperforms Conservative, EASY, and Priority-FIFO on aggregate utility by 53%, 14%, and 6% respectively.

7.2 Scheduler Performance by Decay Type

We have chosen the default behavior of our synthetic utility function generator to be that it generates approximately one-third of the workload using each of the three decay patterns: *expected linear, expected exponential,* and *step*. If one has reason to believe that a particular workload of interest would be more heavily populated by one of these types, it would be useful to know if certain schedulers perform better on that type. Figure 5 shows this comparison using the aggregate utility metric.

Not surprisingly, of the three types of decay models, the *expected exponential* decay model resulted in the lowest score for each algorithm. Jobs simply lose value too quickly to complete all of them before some lose much (or all) of their value. This is also the decay model on which the GA scheduler's performance is highest relative to the other algorithms, an improvement of 40% over the average of the others and 20% over Priority-FIFO.

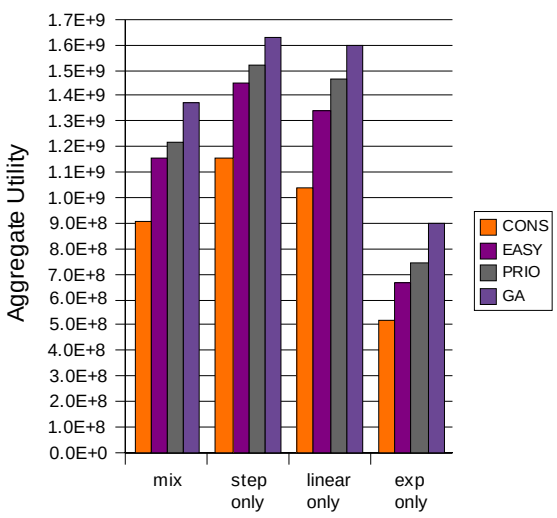


Figure 5. Scheduler performance by decay type.

7.3 Scheduler Performance by System Load

The most difficult test of a scheduler may be its performance in heavy load conditions. When it simply isn't possible to run all jobs in a reasonably timely manner, what should be done? Schedulers perform a balancing act—triage of the importance of each job and how long it has already waited, while keeping an eye on how each action might impact the global good. Perhaps an important job is an awkward "shape" that cannot be fit in without doing an amount of damage to the scheduling of the rest of the jobs that outweighs the benefits.

The SDSC Blue workload already has variation in load, sometimes reaching quite heavy loads, according to local time of day and day of the week (load by time of day shown in Figure 6). We exacerbate this load by scaling the inter-arrival times of jobs in the log by varying factors, as shown in Figure 7.

Figure 6. SDSC Blue's load variation by time of day. Graph from Parallel Workloads Archive. [7] Used with permission.

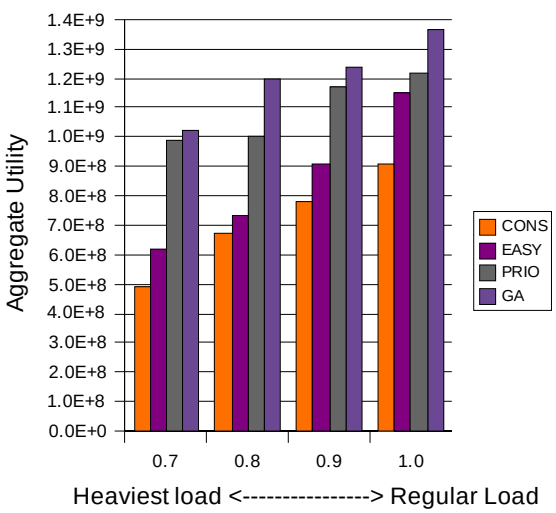


Figure 7. Performance by varying load. To increase load, inter-arrival times are scaled by the factors shown.

Priority-FIFO and GA perform extremely well in high load conditions. The greater the load, the greater the difference in performance between these algorithms and those that do not consider priority or utility. Priority, expressed through a queue choice or a utility function, is a powerful tool to help schedulers manage heavy load.

7.4 Scheduler Performance by Deadline Urgency

Another way to stress a scheduler is to scale the deadlines of the utility functions down, so that jobs are more urgent. Recall that *deadline* is set to be twice the actual wait time for that job recorded in the workload trace. We generate new workloads where this factor is one or three, as shown in Figure 8.

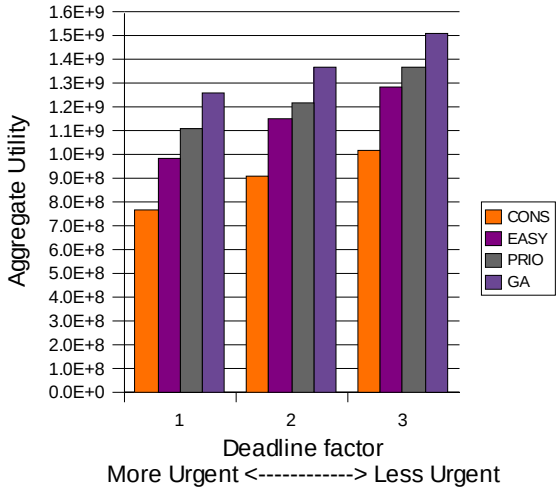


Figure 8. Scheduler performance when varying the workload's utility functions' deadline (larger deadline factor indicates less urgency)

Naturally, all schedulers perform better when they have more time to schedule jobs (*deadline factor* = 3). Under *deadline factor* = 1, EASY proves more resilient with this method of stressing a scheduler than it was when load was increased by scaling interarrival times (Section 7.3, Figure 7). This is probably because one thing EASY does extraordinarily well is aggressively backfilling of small jobs, resulting in a large percentage (even the majority) of jobs having no wait time whatsoever (see Table 1). Thus no matter how soon the deadline of these jobs is moved (as long as it is nonzero) EASY will still earn their full utility.

8 ACCURACY OF REQUESTED RUNTIMES

For the algorithms EASY and Conservative backfilling, there is a well known and yet surprising result that they perform as well or better with inaccurate requested runtimes (relative to actual runtimes) than with completely accurate ones [14]. This result is surprising in that it goes against the "garbage in, garbage out" mantra of computer programming. It was shown [25] that with more and more inaccuracy (larger and larger multipliers on the real runtime), EASY and Conservative degenerate into approximating the Shortest Job First algorithm, which is the optimal algorithm for minimizing average wait in uniprocessor settings. So inaccurate requested runtimes served to transform a less-than-ideal algorithm into a nearly ideal one. But it seems intuitive that an algorithm that is already carefully tuned would still follow the "garbage in" law. (As explained in [21], the original result was also an artifact of an overly "clean" model of user inaccuracy. Actual runtimes were multiplied by the same factor *f* to generate inaccurate ones, instead of using real user estimates, where *f* will of course vary with each job.)

As we explore the aggregate utility metric, and a new algorithm, GA, it is prudent to validate or invalidate the inaccuracy result of [14] as it might apply to these. Table 2 compares schedulers using a variety of metrics, all run using completely accurate requested runtimes (i.e. *requested runtime = actual runtime*).

Table 2. Comparison Using 100% Accurate Requested Runtimes

	Wait Time (s)* 25 %ile 50 %ile 75 %ile 98 %ile 100 %ile	Expansion Factor*† 25 %ile 50 %ile 75 %ile 98 %ile 100 %ile	Average Bounded Slowdown*	Utilization**	Makespan (s)*	% of Job's Max Utility Earned** 25 %ile 50 %ile 75 %ile 98 %ile 100 %ile	Aggregate Utility**
Cons	0 3,621 27,009 246,923 307,491	1.0 1.76 5.28 78.2 393	116	0.468	2.874E+6	73.9 % 0.0198 % 0 % 0 % 0 %	1.03E+09
EASY	11 7,666 35,657 198,790 252,240	1.0 2.00 6.62 99.2 353	164	0.484	2.782E+6	68.2 % 0 % 0 % 0 % 0 %	1.14E+09
Prio	0 383 6,592 159,417 775,694	1.0 1.10 1.98 42.0 861	97	0.485	2.775E+6	89.5 % 30.4 % 0 % 0 % 0 %	1.36E+09
GA	0 140 3,483 381,516 1,153,529	1.0 1.02 1.52 105 1,923	224	0.480	2.801E+6	91.2 % 40.6 % 0.003 % 0 % 0 %	1.41E+09

* Smaller values indicate better performance on these metrics.

** Larger values indicate better performance on these metrics.

† 1.0 is the best possible value for this metric.

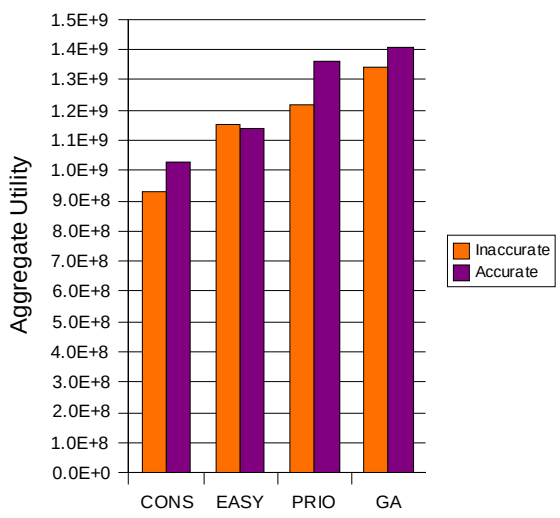


Figure 9. Scheduler performance by accuracy of requested runtimes input to the scheduler. 'Inaccurate' means the actual user-provided requested runtime, 'Accurate' means 100% accurate requested runtimes.

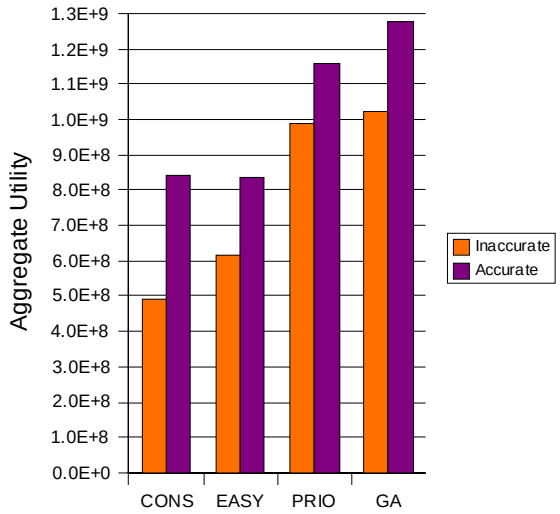


Figure 10. Scheduler performance under extra-heavy load conditions, by accuracy of requested runtimes input to the scheduler. 'Inaccurate' means the actual user-provided requested runtime from the workload trace, 'Accurate' means 100% accurate requested runtimes.

Figure 9 compares each scheduler's performance on the aggregate utility metric when using inaccurate requested runtimes (the actual user-provided requested runtimes), versus using completely accurate requested runtimes. All schedulers except EASY had marked improvement in performance, according to this metric, when using the accurate requested runtimes. EASY's slight decline in performance is consistent with the result when using the *average bounded slowdown* metric.

One hypothesis we wished to explore was if accurate requested runtimes would have a greater positive impact on aggregate utility performance (for those schedulers it does positively impact) under heavy load conditions, than under typical load conditions. Figure 10 compares each scheduler's performance when using inaccurate requested runtimes, versus using completely accurate requested runtimes, on the heavy-load workload used in Figure 7 (using a 0.7 interarrival time factor, see description in Section 7.3). Notice that Priority-FIFO performed nearly equal to GA when both were using the user-provided requested runtimes. Both improved with accurate information, however, GA was able to make better use of the information, improving more than Priority-FIFO did. It appears that under heavy load, accurate information is important to GA.

9 CONCLUSION

Whole user happiness is no doubt a function of more variables than just turnaround time of jobs. It could include for example, predictability of wait times, friendliness of staff, ease of use of storage systems, availability of debuggers and other auxiliary tools, and so on. This work focused on a form of satisfaction our previous work showed users are willing and able to quantify. We presented evidence that overly simplistic formulations cannot capture the nuances of real users' preferences and exhibited a method that can capture those nuances without being overly complicated. We provide a method that scheduling researchers can use to generate realistic utility functions for the purposes of augmenting job logs. And we used the method ourselves to evaluate, via simulation, several classic scheduling algorithms, and a new one based on genetic algorithms, for their ability to improve aggregate utility. Priority-FIFO, with a crude approximation of user utility, outperforms Conservative and EASY flavors of backfilling by 18.4% average under realistic conditions. The GA approach outperforms Priority FIFO by an additional 12.6% average, while consuming short enough computation time to allow it to run in a real-time environment.

10 ACKNOWLEDGEMENTS

This work was supported in part by the DOE SciDAC2 award entitled the Performance Evaluation Research Center (PERC) and by the DOD High Performance Computing Modernization Office. This work was also supported in part by NSF Award #0516162 entitled The NSF Cyberinfrastructure Evaluation Center.

11 REFERENCES

[1] AuYoung, Alvin, Laura Grit, Janet Wiener and John Wilkes. “Service contracts and aggregate utility functions.” *15th IEEE International Symposium on High Performance Distributed Computing*, Paris, France, June 2006.

[2] Chun, Brent. N. Market-based Cluster Resource Management. PhD thesis. University of California, Berkeley. November 2001.

[3] Chun, Brent N., Philip Buonadonna, Alvin AuYoung, Chaki Ng, David C. Parkes, Jeffrey Shneidman, Alex

C. Snoeren, and Amin Vahdat. “Mirage: A Microeconomic Resource Allocation System for Sensornet Testbeds,” *2nd IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, Sydney, Austrailia, May 2005.

[4] Chun, Brent. N. and David E. Culler. “User-centric performance analysis of market-based cluster batch schedulers.” *2nd IEEE International Symposium on Cluster Computing and the Grid*, May 2002.

[5] Chapin, Steve J., Walfredo Cirne, Dror G. Feitelson, James Patton Jones, Scott T. Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. “Benchmarks and Standards for the Evaluation of Parallel Job Schedulers.” In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph eds. 1999.

[6] Earheart, Travis and Nancy Wilkins-Diehr of SDSC provided the original workload to the Parallel Workloads Archive [7, 5] of Dror Feitelson et al. The specific Parallel Workoads Archive file version we used is SDSC-BLUE-2000-2.1-cln.swf. (Version 2.1 has a more streamlined representation of jobs’ priorities than version 3.1.)

[7] Feitelson, Dror, et al. Parallel Workloads Archive and Standard Workload Format. http://www.cs.huji.ac.il/labs/parallel/workload/.

[8] Feitelson, Dror G. and Larry Rudolph. “Toward convergence in job schedulers in parallel supercomputers,” *Proc. of the 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, eds. April 1996.

[9] Feitelson, Dror G., Larry Rudolph, Uwe Schwiegelsohn, Kenneth C. Sevcik and Parkson Wong. “Theory and Practice in Parallel Job Scheduling.” *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, eds. 1997.

[10] Irwin, David E., Laura E. Grit, Jeffrey S. Chase. “Balancing Risk and Reward in a Market-Based Task Service,” *13th IEEE International Symposium on High Performance Distributed Computing*, June 2004.

[11] Lai, K. B. A. Huberman and L. Fine. “Tycoon: A Distributed Market-based Resource Allocation System. Technical Report.” cs.DC/0404013, April 2004. Available at http://arxiv.org/abs/cs.DC/0404013

[12] Lee, Cynthia Bailey and Allan Snavely. “On the User-Scheduler Dialogue: Studies of User-Provided Runtime Estimates and Utility Functions.” *International Journal of High Performance Computing Applications*, 2006.

[13] Lee, Cynthia Bailey, Yael Schwartzman, Jennifer Hardy and Allan Snavely “Are user runtime estimates inherently inaccurate?” In *10th Job Scheduling Strategies for Parallel Processing*, June 2004.

[14] Mu'alem, Ahuva W. and Dror G. Feitelson. “Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling,” *IEEE Trans. Parallel & Distributed Systems*, 12(6). June 2001.

[15] Norvig, Peter. Python code, available at http://aima.cs.berkeley.edu/python/readme.html. Used according to terms of license. © 1998 – 2002.

[16] Pfeiffer, Wayne. Personal Interview. San Diego Supercomputer Center, at the University of California, San Diego. La Jolla, CA. April 9, 2004.

[17] Russell, Stuart and Peter Norvig. *Artificial Intelligence: A Modern Approach*, Prentice Hall Series in Artificial Intelligence. Englewood Cliffs, New Jersey. 1995.

[18] Stoica, Ion, Hussein Abdel-Wahab and Alex Pothen. “A Microeconomic Scheduler for Parallel Computers,” *Proc. of the 1st Workshop on Job Scheduling Strategies for Parallel Processing,* D. G. Feitelson and L. Rudolph, eds. April 1995.

[19] Talby, David, Dror G. Feitelson and A. Raveh. “Comparing logs and models of parallel workloads using the coplot method.” *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (Eds.), 1999.

[20] Tsafrir, Dan and Dror G. Feitelson. “Instability in parallel job scheduling simulation: the role of workload flurries.” *IEEE International Parallel and Distributed Processing Symposium.* Rhodes Island, Greece, April 2006.

[21] Tsafrir, Dan and Dror G. Feitelson. “The dynamics of backfilling: solving the mystery of why increased inaccuracy may help.” *IEEE International Symposium on Workload Characterization.* San Jose, California. October 2006.

[22] Waldspurger, C. A., T. Hogg, B. A. Huberman, J. O. Kephart and S. Stornetta. “Spawn: A Distributed Computational Economy.” *IEEE Transactions on Software Engineering*, 18(2). February 1992.

[23] Weinberg, Jon and Allan Snavely. “Symbiotic Space-Sharing on SDSC's DataStar System.” *Proceedings of the 12th Workshop on Job Scheduling Strategies for Parallel Processing,* E. Frachtenberg and U. Schwiegelshohn, eds. 2006.

[24] Wolski, Rich, James Plank, John Brevik and Todd Bryan. “Analyzing Market-based Resource Allocation Strategies for the Computational Grid.” *The International Journal of High Performance Computing Applications, Vol. 15: 3*. 2001.

[25] Zotkin, Dmitry and Peter J. Keleher. “Job-length estimation and performance in backfilling schedulers.” *8th High Performance Distributed Computing Conference. IEEE*. 1999.