A Reinforcement Learning Framework for Utility-Based Scheduling in Resource-Constrained Systems*

David Vengerov
Sun Microsystems Laboratories
UMPK16-160
16 Network Circle
Menlo Park, CA 94025
1-650-520-7157
david.vengerov@sun.com

Abstract

This paper presents a general methodology for online scheduling of parallel jobs onto multi-processor servers in a soft real-time environment, where the final utility of each job decreases with the job completion time. A solution approach is presented where each server uses Reinforcement Learning for tuning its own value function, which predicts the average future utility per time step obtained from completed jobs based on the dynamically observed state information. The server then selects jobs from its job queue, possibly preempting some currently running jobs and "squeezing" some jobs into fewer CPUs than they ideally require, so as to maximize the value of the resulting server state. The experimental results demonstrate the feasibility and benefits of the proposed approach.

Keywords: Reinforcement Learning, Utility Computing, Job Scheduling, Utilization, Time Utility Functions

*This material is based upon work supported by DARPA under Contract No. NBCH3039002.

1 Introduction

The classical real-time scheduling problem is that of deciding on the order in which the currently unscheduled jobs should be executed by a single processor, with no possibility of processor sharing. In the case of *hard real-time systems*, each job is characterized by an execution time and a deadline. If such a system is underloaded (it is possible to schedule all incoming jobs without missing any deadlines), then the Earliest Deadline First (EDF) algorithm [12] will construct a schedule with no missed deadlines.

A *soft real-time system* is the one where each job receives some utility as a function of its completion time. The concept of scheduling jobs based on Time Utility Functions (TUFs) was first introduced by Jensen in [5], where the objective of maximizing the total utility accrued by the system over time was used. The utility accrual (UA) paradigm is a generalization of the deadline scheduling in hard real-time systems. That is, if the system receives a utility of 1 for completing a job before its deadline and a utility of 0 otherwise, then the EDF algorithm will maximize the total utility accrued by the system if it is possible to complete all jobs before their deadlines. An important benefit of the UA paradigm is that it allows one to optimize the *productivity* of the system: average utility from completed jobs per unit of time [8, 9].

However, as the average load in the system increases and some jobs become impossible to complete before their deadlines, then performance of EDF degrades rapidly [14] in terms of the total utility accrued by the system. The recent overviews of this field [14, 21] document three "standard" algorithms that can handle the case of overloaded systems within the UA paradigm: DASA [3], LBESA [5], and Dynamic Priority [15]. Performance of these algorithms was found to be very similar for a broad range of workloads and TUFs. Variants of DASA and LBESA have been implemented in the Alpha real-time operating system [4] and the MK7.3 kernel [16].

While the above algorithms might be reasonable heuristics for single-processor scheduling, they cannot handle the case of scheduling multi-CPU jobs onto multi-processor systems, which arises in massively parallel computing systems, Grid systems and data centers. Recently, the originator of the UA paradigm E. D. Jensen has co-authored a paper [21], which claims to present the only scheduling al-

gorithm that can handle arbitrarily shaped TUFs and multi-unit resource constraints (multiple resource types, each having multiple units). The authors also claim that their earlier algorithm [10, 11] is the only algorithm that can handle arbitrarily shaped TUFs and single-unit resource constraints (one type of resource with multiple units). These algorithms are very similar and can both apply to the multi-processor scheduling problem considered in this paper. These algorithms are based on the idea that since the future is unpredictable, it is best to use the "greedy" strategy of inserting as many "high utility" jobs into the schedule as early as possible. While it is a reasonable heuristic for a resource-constrained environment, it is possible to develop algorithms that do try to predict the future and hence can perform better than the greedy strategy.

The original paradigm of controlling a system's behavior (scheduling jobs onto the system's resources, etc.) based on a model for evolution of the system's state was developed within the Control Theory. The objective of Control Theory is to apply the controls so as to keep a certain observable system characteristic within the desired bounds. Some attempts have been made to apply the Control Theory paradigm to develop schedulers for hard real-time systems (e.g., [13]), with the dependent parameter being the fraction of jobs that miss their deadlines.

The formulation of classical Control Theory cannot be used to address the scheduling problem within the UA paradigm, which has the long-term performance objective of maximizing the accrued utility. This objective can be handled by the more recently developed Optimal Control Theory. The discrete-time optimal control problem formulation is also known as the Markov Decision Process (MDP), which can be solved using Dynamic Programming [1] if the system model is known and the number of possible system states is small. Neither of these conditions is satisfied in real-world systems of interest, and the Reinforcement Learning methodology [2, 7, 17] was developed as a formal extension of dynamic programming to the case of unknown system models and large state spaces. We are not aware of any attempts of using the Optimal Control or the MDP formulation for solving the real-time scheduling problem within the UA paradigm, for a single CPU or a multi-CPU environment.

This paper addresses the above shortcomings by presenting a general MDP framework and algorithms for solving scheduling problems within the UA paradigm in the multi-CPU environment. The key fea-

ture of the proposed scheduling approach is approximation of the value function that gives expected long-term productivity of each machine (average utility per time step obtained from all completed jobs) as a function of its current state, which is accomplished using the powerful Reinforcement Learning methodology. Once an approximation to this value function is obtained, any scheduling decisions can be made that modify the existing state of each machine with the goal of increasing its state value. As an example, we extend the traditional scheduling problem (where one needs to decide only on the order in which the jobs are scheduled) with two additional scheduling decisions: deciding whether or not some of the currently running jobs should be preempted in order to fit a given unscheduled job, and deciding whether or not an unscheduled job should be "squeezed" onto a machine that has fewer remaining CPUs than desired by the job.

The experimental results implementing the above utility-based scheduling approach are presented, which demonstrate a consistent increase in machine productivity over the standard scheduling approach in the UA domain. This benchmark approach is the one lying at the heart of the algorithms presented in [3, 10, 11, 21], suitably extended to handle multi-CPU jobs contending for a limited number of CPUs.

2 Problem Formulation

Each job arriving into the system requires a certain number of CPUs to be executed at the maximum rate but can be executed with fewer CPUs at a slower rate. The final job utility decays as its waiting+execution time increases. In addition to deciding on the order in which the arriving jobs should be scheduled, the scheduler is also allowed to "squeeze" a job into a smaller number of CPUs than it ideally requires, thereby extending its execution time and receiving a smaller final utility. Alternatively, the scheduler can wait until the initially requested number of CPUs becomes available for a particular job in order to ensure the maximum execution rate, while risking to have a long waiting time for this job and a low final utility once again. The scheduler can also suspend some of the currently running jobs so as to schedule some of the waiting jobs, and resume the execution of the suspended jobs at a later time. If some CPUs become available before the "squeezed" job completes its execution, the scheduler

assigns more CPUs to that job until it gets all its originally requested CPUs.

This paper presents a novel utility-based scheduling framework capable of adequately resolving the tradeoffs mentioned above. An overview of the considered problem is given below. More details will be given in section 5.

The *unit* utility of each job is a decreasing function of the job completion time, which includes the job waiting time. The *final* utility of each job is its unit utility multiplied by K*L, where K is its desired number of CPUs and L is its *ideal* execution time if the job were to receive K CPUs. The K*L factor is introduced to reflect the assumption that larger and longer jobs should receive the same scheduling priority as smaller and shorter jobs – to make the scheduler indifferent between scheduling one job requiring K CPUs and having the ideal execution time L and scheduling K*L jobs each of which requires one CPU and has an ideal execution time of one unit of time.

The jobs are assumed to arrive randomly into the system. The goal is to continually schedule incoming jobs onto the system and to decide how many CPUs should be allocated to each job so as to maximize average system productivity (final utility from all completed jobs) per time step.

3 Solution Methodology

3.1 Overview

The proposed utility-based scheduling framework most generally applies to the multi-machine grid environment (or a multi-module massively parallel computer), where multiple machines may be available to accept multiple waiting jobs. The basic scheduling algorithm uses the best-fit technique: each machine selects the next job to be scheduled which has the tightest fit for this machine (will have the fewest free CPUs remaining after the job is scheduled). If there is a tie among the jobs that provide the best fit to the available resources, then the highest-utility job gets scheduled. In addition to this basic scheduling algorithm, this paper considers two additional types of decisions that can be made by the scheduler independently on each machine, which apply if the machine cannot fit any of the currently waiting jobs.

The first *preemption* policy decides whether one or several of the jobs currently running on that machine should be temporarily suspended in order to allow one of the waiting jobs to be scheduled with all the desired CPUs. The suspended jobs are placed in the waiting queue for that machine and can be resumed from the place they left off. The second *oversubscribing* policy decides whether any of the currently waiting jobs should be "squeezed" into the remaining free CPUs on that machine even if there are fewer of them than the job ideally desires.

Both of these policies make scheduling decisions by selecting the job configuration that provides the best starting point for the machine's future operations in terms of maximizing the *long-term utility* (value) from all jobs completed in the future. Thus, the key component of the proposed scheduling framework is the possibility of learning the value function for each machine based on the state of its resources, the jobs it is executing and the jobs waiting to be scheduled. Once such a value function is obtained, any kind of scheduling decisions can be considered on each machine (not just preemption and oversubscribing), which can now be driven by maximization of the machine's state value following the scheduling decision.

The most common approach for learning functions that match inputs to outputs is to assume some flexible function approximation architecture with tunable parameters and then adjust these parameters so as to improve the quality of approximation. Any kind of parameterized value function approximation architectures can be used in the proposed scheduling framework, since the only criterion for its compatibility with the reinforcement learning methodology is function differentiability with respect to each tunable parameter. As a demonstration, a parameterized fuzzy rulebase will be used in this paper to approximate machine value functions. The fuzzy rulebase parameters will then be tuned using the reinforcement learning process described in Section 3.4, which consists of observing the utilities of completed jobs following different states of the machine and changing parameters of the value function so as to increase the value of states after which high job utilities were observed while lowering the value of the states after which low job utilities were observed.

3.2 Fuzzy Rulebase

A fuzzy rulebase is a function f that maps an input vector $x \in \Re^K$ into a scalar output y. This function is formed out of fuzzy rules, where a fuzzy rule i is a function f_i that maps an input vector $x \in \Re^K$ into a scalar p^i. The following common form of the fuzzy rules is used in this paper:

Rule i: IF (x_1 is S_1^i) and (x_2 is S_2^i) and ... (x_K is S_K^i) THEN (output=p^i),

where x_j is the jth component of x, S_j^i are the input labels in rule i and p^i are the output coefficients. The degree to which the linguistic expression (x_j is S_j^i) is satisfied is given by a *membership function* $\mu : \Re \rightarrow \Re$, which maps its input x_j into the degree to which this input belongs to the fuzzy category described by the corresponding label. The output of the fuzzy rulebase $f(x)$ is a weighted average of p^i:

$$y = f(x) = \frac{\sum_{i=1}^{M} p^i w^i(x)}{\sum_{i=1}^{M} w^i(x)}, \tag{1}$$

where M is the number of rules and $w^i(x)$ is the weight of rule i. The product inference is commonly used for computing the weight of each rule: $w^i(x) = \prod_{j=1}^{K} \mu_{S_j^i}(x_j)$. The membership functions $\mu_{S_j^i}(x_j)$, $i = 1, ..., M$, are usually chosen so as to jointly cover the range of possible values of the input variable x_j. Therefore, each fuzzy rule can be visualized as a box in space with fuzzy boundaries (jointly these boxes cover the range of possible values for x), so that when x is observed within a box i, the output recommended by the rule i is p^i. The actual rulebase output is a weighted sum of p^i, with the weights indicating the "distance" between x and the center of box i. If the membership functions $\mu()$ are kept constant and only the output coefficients p^i are tuned, then the fuzzy rulebase becomes equivalent to a linear combination of basis functions – a well known statistical regression model. If the membership functions are tuned as well, then the above form of the fuzzy rulebase was proven to be a universal function approximator [19], just like a multi-layer perceptron neural network. However, tuning the membership functions requires a nonlinear learning algorithm, which is much harder to set up and use. In practice, tuning the output coefficients p^i is often sufficient to come up with a good policy, and for simplicity of exposition this approach is taken in this paper.

3.3 Value Function Architecture for Job Scheduling

As a simple demonstration of the proposed scheduling framework, we use the following variables for computing the *value* of machine state at any point in time (a more sophisticated input set could be considered):

- x_1 = average unit utility expected to be received by the currently running jobs (introduced in Section 2 and computed from Figure 1) weighted by the number of CPUs each job is currently occupying,
- x_2 = the expected time remaining until any of the currently running jobs is completed,
- x_3 = the number of currently idle CPUs on the machine.

Our experimental results showed that the three input variables described above are sufficient for encoding the machine state under the assumption that "squeezing" a job on a given machine affects only the execution time of that job. If, however, the execution time of all jobs on that machine is affected, then a fourth variable should be used by each machine – the average degree to which all of its jobs are "squeezed."

The above variables are used as inputs to the fuzzy rulebase, and its output $\hat{V}(x)$ is an approximation to the expected future utility per time step obtained from completed jobs when starting from the state x. A description of the reinforcement learning algorithm for updating the rulebase parameters p^i to improve its approximation quality will be given in section 3.4.

The following fuzzy rulebase were used to represent $\hat{V}(x)$:

Rule 1: IF (x_1 is S_1) and (x_2 is S_2) and (x_3 is S_3) then p^1

Rule 2: IF (x_1 is S_1) and (x_2 is S_2) and (x_3 is L_3) then p^2

Rule 3: IF (x_1 is S_1) and (x_2 is L_2) and (x_3 is S_3) then p^3

Rule 4: IF (x_1 is S_1) and (x_2 is L_2) and (x_3 is L_3) then p^4

Rule 5: IF (x_1 is L_1) and (x_2 is S_2) and (x_3 is S_3) then p^5

Rule 6: IF (x_1 is L_1) and (x_2 is S_2) and (x_3 is L_3) then p^6

Rule 7: IF (x_1 is L_1) and (x_2 is L_2) and (x_3 is S_3) then p^7

Rule 8: IF (x_1 is L_1) and (x_2 is L_2) and (x_3 is L_3) then p^8

The above rules softly classify each input variable x_j into two fuzzy categories: small (S) and large (L). The weight w^i of the fuzzy rule i is the product of the degrees to which every precondition is satisfied. The membership functions μ, which compute these degrees, have the following form:

- degree to which (x_1 is L_1): $\mu_{L_1}(x_1) = x_1$
- degree to which (x_2 is L_2): $\mu_{L_1}(x_2) = x_2/MaxJobLength$
- degree to which (x_3 is L_3): $\mu_{L_3}(x_3) = x_3/N$,
- degree to which (x_1 is S_1): $\mu_{S_1}(x_1) = 1 - x_1$
- degree to which (x_2 is S_2): $\mu_{S_2}(x_2) = 1 - x_2/MaxJobLength$
- degree to which (x_3 is S_3): $\mu_{S_3}(x_3) = 1 - x_3/N$

where N is the maximum number of CPUs on the machine and $MaxJobLength$ is the maximum length of a job that can be scheduled on this machine. The output of the above fuzzy rulebase is computed according to equation (1).

Since each membership function $\mu(x_j)$ is continuous and the product inference is used, the output of the fuzzy rulebase changes smoothly as the inputs change, conforming to the idea described in section 3.4 of "softly" generalizing the learned experience across similar states. The rulebase parameters p^i can be set using prior knowledge or dynamically adjusted starting from any initial values using the reinforcement learning (RL) procedure described in the next section.

3.4 Reinforcement Learning Algorithm for Tuning Value Functions

We first describe the general mathematical context of the Markov Decision Process (MDP) where reinforcement learning (RL) algorithms can be used. An MDP for a single agent (decision maker) can be described by a quadruple (S, A, R, T) consisting of:

- A finite set of states S
- A finite set of actions A
- A reward function $r : S \times A \times S \rightarrow \Re$
- A state transition function $T : S \times A \rightarrow PD(S)$, which maps the agent's current state and action into the set of probability distributions over S.

At each time t, the agent observes the state $s_t \in S$ of the system, selects an action $a \in A$ and the system changes its state according to the probability distribution specified by T, which depends only on s_t and a_t. The agent then receives a real-valued reward signal $r(s_t, a_t, s_{t+1})$. The agent's objective is to find a stationary policy $\pi : S \rightarrow A$ that maximizes expectation the average reward per time step starting from any initial state.

For a stationary policy π, the average reward per time step is defined as:

$$\rho^{\pi} = \lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=0}^{T-1} r(s_t, \pi(s_t), s_{t+1}), \tag{2}$$

The optimal policy π^* from a class of policies Π is defined as $\pi^* = \underset{\pi \in \Pi}{\text{argmax}} \rho^{\pi}$. The *value* of a state s under a policy π is defined as:

$$V^{\pi}(s) = E[\sum_{t=0}^{\infty} (r(s_t, \pi(s_t), s_{t+1}) - \rho^{\pi}) | s_0 = s]. \tag{3}$$

A well-known procedure for iteratively approximating $V^{\pi}(s)$ is called *temporal difference* (TD) learning. Its simplest form is called TD(0):

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha_t (r(s_t, \pi(s_t), s_{t+1}) - \rho_t + \hat{V}(s_{t+1}) - \hat{V}(s_t)), \tag{4}$$

where α_t is the learning rate, $\hat{V}(s)$ is the current approximation to $V^\pi(s)$ when the policy π is being followed and ρ_t is updated as:

$$\rho_t = (1 - \alpha_t)\rho_t + \alpha_t r(s_t, \pi(s_t), s_{t+1}). \tag{5}$$

The above iterative procedure converges as long as the underlying Markov chain of states encountered under policy π is irreducible and aperiodic and the learning rate α_t satisfies $\sum_{t=0}^{\infty} \alpha_t = \infty$ and $\sum_{t=0}^{\infty} \alpha_t^2 < \infty$ [18]. For example, $\alpha_t = 1/t$ satisfies this conditions.

The TD approach based on assigning a value to each state becomes impractical when the state space becomes very large or continuous, since visits to any given state become very improbable. In this case, a function approximation architecture needs to be used in order to generalize the value function across neighboring states. Let $\hat{V}(s, p)$ be an approximation to the value function $V^\pi(s)$ based on a linear combination of basis functions with a parameter vector p: $\hat{V}(s, p) = \sum_{i=1}^{M} p^i \phi^i(s)$. The parameter updating rule in this case becomes (executed for all parameters simultaneously):

$$\begin{aligned} p_{t+1}^i &= p_t^i + \alpha_t \frac{\partial}{\partial p^i}[r(s_t, \pi(s_t), s_{t+1}) - \rho_t + \hat{V}(s_{t+1}, p_t) - \hat{V}(s_t, p_t)]^2 \\ &= p_t^i + \alpha_t[r(s_t, \pi(s_t), s_{t+1}) - \rho_t + \hat{V}(s_{t+1}, p_t) - \hat{V}(s_t, p_t)] \frac{\partial}{\partial p^i} \hat{V}(s_t, p_t) \\ &= p_t^i + \alpha_t[r(s_t, \pi(s_t), s_{t+1}) - \rho_t + \hat{V}(s_{t+1}, p_t) - \hat{V}(s_t, p_t)] \phi^i(s_t), \end{aligned} \tag{6}$$

where the average reward estimate ρ_t is updated as in equation (5). The above iterative procedure is also guaranteed to converge to the locally optimal parameter vector p^* (the one giving the best approximation to $V^\pi(s)$ in its neighborhood) if certain additional conditions are satisfied [18]. The most important ones are that the basis functions $\phi^i(s)$ are linearly independent and that the states for update are sampled according to the steady-state distribution of the underlying Markov chain for the given policy π.

Note that the state description chosen for each machine in section 3.3 implies that a value-maximizing scheduling decision immediately changes the machine state s to a higher-valued state $\tilde{s}$, from which the state evolution proceeds. The particular RL methodology we propose for tuning the value function

parameters makes use of this observation by specifying that the state s_t in equation (6) is the one observed AFTER a scheduling decision (if any) is taken. That is, decisions made during the scheduling process affect only the way the states are sampled for update in equation (6), but the evolution of a state s_t to a state s_{t+1} is always governed by a transition policy π embedded into the particular scheduling environment (which depends on the job arrival rate, on the distribution of job characteristics, etc.). So far no theoretical convergence analysis has been performed for the proposed sampling approach of choosing at every decision point the highest-valued state out of those that can be obtained, but our positive experimental results suggest its feasibility.

Also note that the above view of the learning process as that of estimating the value function for a fixed state transition policy eliminates the need for performing action exploration (which is necessary when we try to learn a new state transition policy), thus making it possible to use this approach on-line without degrading performance of the currently used policy. For example, if we initialize the machine value function to prefer some obviously bad states, then after some period of learning and tuning the value function parameters, the values of these states will come down to their real levels and the system will start using appropriate state values when making scheduling decisions, without an explicit exploration having been performed.

The final defining aspect of the RL algorithm we propose for the scheduling domain described in Section 2 is the timing of the value function parameter updates. The traditional implementation of RL is to update parameters whenever the system's state changes. However, in our case this would lead to parameter updates at every time step, since the input variable 2 to the value function (the expected time remaining until any of the currently running jobs is completed) constantly changes. Such frequent parameter updates would "dilute" the impact of any action taken by the scheduler (such as preemption/squeezing) on the future states and also create an unnecessary computational load on the scheduler. Instead, we propose to perform the updates using equation (6) when the machine state is changed due to a job being preempted or "squeezed" into fewer than desired CPUs. The parameters are also updated when the machine state is changed due to a job completing or a new job being scheduled following the situation when the decision to "squeeze" or preempt any jobs was considered but not made. In this way,

the connection between the most recent "free action" by the scheduler and the next state when another action can be taken is most clear.

A fuzzy rulebase, as described in section 3.2, is an instance of a linear parameterized function approximation architecture, where the normalized weight $\frac{w^i(s)}{\sum_{i=1}^{M} w^i(s)}$ of each rule i is a basis function $\phi^i(s)$ and p^i are the tunable parameters. The reward signal $r(s_t, \pi s_t, s_{t+1})$ is computed as the total utility received from completed jobs during the time elapsed between states s_t and s_{t+1}. The vector x of section 3.3, used as an input to the fuzzy rulebase, is treated as the state vector s in equation (6) when updating the parameters of the fuzzy rulebase. The vector x does not summarize all the information needed for determining its expected future evolution, which implies that we have set up this problem as a Partially Observable MDP (POMDP) [6]. This was a conscious decision, since accounting for all the relevant information (the expected completion time of each running job, the number of CPUs each job occupies and the characteristics of all currently waiting jobs) would lead to a very high-dimensional state, making the learning very difficult. The experimental results presented in the next section demonstrate that the RL approach presented above is robust enough to learn good decision policies starting from parameters p^i are being initialized at 0 even when the system dynamics deviates from the ideal MDP environment (when the state summarizes all relevant information and decisions can be made at every time step).

4 Value-Based Job Scheduling Algorithm

4.1 Scheduling on a Single Machine

Given an architecture for approximating the machine state value (which can still be in the process of being tuned by RL), the following general steps are taken by the scheduling algorithm whenever a new job arrives into the queue or one of the currently running jobs is completed:

1. Schedule jobs onto the free CPUs without any preemption or oversubscribing using any "traditional" scheduling approach such as best-fit scheduling.

If any jobs are still waiting, perform the following steps:

2. Compute the variables x_1, x_2, and x_3 for the current machine state.

3. Use the computed state vector x as an input to the fuzzy rulebase to compute V_0 – the long-term expected utility (average utility per time step) that can be obtained by the machine if none of the currently waiting jobs are scheduled and the same algorithm is invoked at every future decision point.

If the preemption policy is enabled, the scheduling algorithm performs the following two steps for every waiting job i:

4. Compute the alternate possible machine state in terms of the new values for x_1, x_2, and x_3 that would arise if job i were forcefully scheduled, preempting enough jobs to fit itself, in the order of increasing *remaining utility*. The remaining utility of each running job is its expected unit utility at completion divided by the expected time remaining to completion. If job i needs to preempt a job with a higher remaining unit utility than its own in order to fit itself (e.g., if all the currently running jobs have high remaining unit utilities), then it is eliminated from further consideration for forceful scheduling.

5. If job i qualifies for forceful scheduling, use the alternate state vector x' computed in step 4 as an input to the fuzzy rulebase to compute V_1^i – the long-term expected utility that can be obtained by the machine if job i were forcefully scheduled, assuming the same algorithm will be invoked at every future decision point.

6. Let $MaxV1 = \max_i(V_1^i)$

7. If $MaxV1 > V_0$ then preempt enough lowest-utility jobs and schedule the job with the highest V_1^i onto the machine; otherwise, do not preempt any jobs and do not schedule any of the currently waiting jobs.

If the oversubscribing policy is also enabled, the scheduling algorithm performs the following two steps for every job i that is still waiting to be scheduled:

8. Compute the alternate machine state in terms of the variables x_1, x_2, and x_3 that would result if job i were scheduled onto the free CPUs, oversubscribing the machine.

9. Use the alternate state vector x' computed in step 8 as an input to the fuzzy rulebase to compute V_2^i – the expected long-term benefit of oversubscribing the machine with the job i, assuming the same algorithm will be invoked at every future decision point.

10. Let $MaxV2 = \max_i(V_2^i)$

11. If $MaxV2 > V_0$ then oversubscribe the machine with the job that has the highest V_2^i; otherwise - do not oversubscribe.

4.2 Scheduling on Multiple Machines

If several machines are available to accept the waiting jobs, then the scheduling algorithm needs to decide how to allocate jobs among the machines. The natural objective function for this problem is to maximize the sum of values of all machines at the end of the allocation process. Unfortunately, the classical allocation algorithms based on Integer Programming do not apply to this problem because the value of each machine is nonlinear in the jobs allocated to it. However, various heuristics can be used instead, where a given allocation is perturbed in some way, and the new allocation is implemented if it increases the total value of all machines. A study of various job allocation heuristics and allocation perturbation methods on multiple machines is outside the scope of this paper, as we want to demonstrate here the possibility of learning machine value functions that are required for all such heuristics.

As a simple example, following two-stage process can be used. First, jobs are scheduled without preemption or oversubscribing until no more jobs can fit in. This can be accomplished by scheduling jobs one at a time by fixing a machine and finding the best-fitting job or fixing a job and finding the best-fitting machine. The first heuristic is expected to work better when the number of machines is small relative to the number of unscheduled jobs, while the second one is expected to work better in the opposite scenario. In the second stage of the scheduling process, each machine needs to decide whether some of the currently running jobs should be preempted in order to schedule any jobs that are still waiting or "squeeze" any of the currently waiting jobs into the CPUs that are still available. In the simplest case, each machine can independently execute the algorithm described in section 4.1.

5 Simulation Results

Performance of adaptive utility-based policies for oversubscribing and preemption was evaluated using a Grid simulator developed in collaboration with the DReAM team at the Sunnyvale Ranch of Sun Microsystems. The test results demonstrated a consistent increase in machine productivity over non-

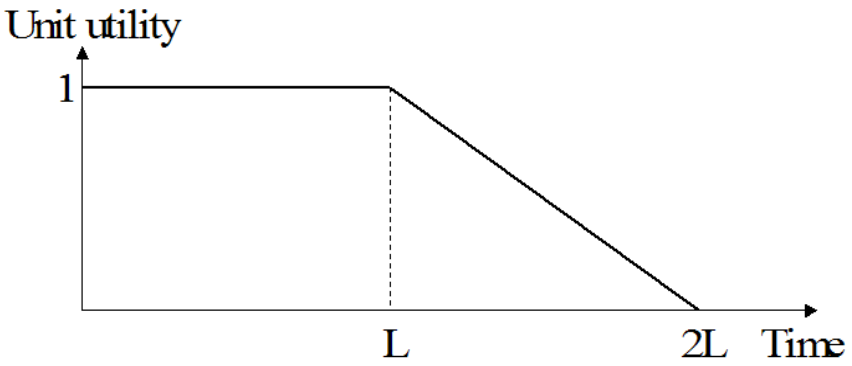

Figure 1. Time Utility Function (TUF) used by the jobs, where L is the ideal execution time.

adaptive policies when either one of the utility-based policies (job preemption or "squeezing") was deployed and adaptively tuned using the RL methodology, with the maximum productivity obtained when both policies were deployed and tuned simultaneously. The experimental results are summarized in Figures 2-5. They were obtained for the "base case" for studying the feasibility of learning good machine value functions – one machine with 24 CPUs, the maximum machine size currently used in the Sunnyvale Ranch. The number of CPUs required for each job was sampled from a random uniform distribution on [4, 20] and the ideal job execution time was sampled from a random uniform distribution on [5, 10]. The unit utility of each job as a function of the total completion time (TUF) was equal to 1 between t = 0 and t = "ideal execution time L" and then decayed to 0 at t = 2L, as shown in Figure 1. Jobs arrived stochastically in a Poisson manner. For simplicity, we assumed that the job execution time is given by $L\frac{N_I}{\min(N_A,N_I)}$, where N_I is the ideal number of CPUs requested by the job, L is the ideal execution time if the job were given N_I CPUs and N_A is the number of CPUs actually allocated to the job. Any other dependence on the number of assigned CPUs could have been used, since our scheduling algorithms do not use this information explicitly. All results are given over the test period of 20000 time steps and are averaged over 50 experiments, so that the standard deviation of each performance observation is less than 1% of the observation itself.

Figure 2 shows performance of various scheduling policies for the job arrival rate of 0.1, evaluating the machine productivity (total final utility of all completed jobs), in terms of 1000 utility units. The first row corresponds to the best-fit scheduling policy with no preemption or oversubscribing. Whenever there is

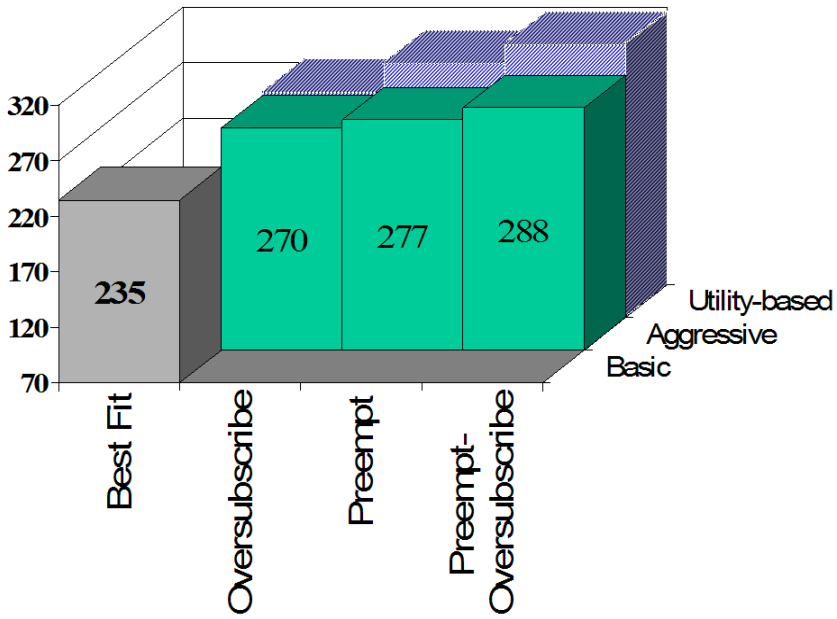

Figure 2. Machine productivity for various policies.

a tie among jobs that provide the best fit to the available CPUs, the job with the highest utility upon completion gets scheduled first, in the spirit of the best known scheduling algorithms [3, 10, 11, 21]. This policy presents one side of the spectrum of possible oversubscribing/preemption policies, where such decisions are never made.

On the other hand, the second row corresponds to the strategies of "aggressive" oversubscribing, preemption, and combined preemption-oversubscribing policies. Each of these policies uses best-fit scheduling as described above, and if no jobs can be scheduled during some time step but some jobs are waiting, a basic policy oversubscribes the machine by scheduling a job with fewer than desired CPUs, tries to forcefully schedule a job (preempting, if possible, some of the lower-valued currently running jobs as explained in section 4.1), or considers preemptive scheduling followed by oversubscribing, depending on the exact policy used. The smallest of the waiting jobs is scheduled by an "aggressive" policy, which showed better results than scheduling the highest-utility job (which would get "squeezed" to a larger extent or would preempt more jobs).

The third row corresponds to the utility-based oversubscribing, preemption, and combined preemption-

oversubscribing policies. These policies made their decisions based on maximizing the value of machine state as explained in section 4.1. Prior to testing a utility-based policy, parameters of the utility function approximation architecture $\hat{V}(x)$ were adjusted for 25000 time steps using the reinforcement learning equation (6) starting from $p^i = 0$. As an example, the values below define the value function learned by the preemption-oversubscribing policy in one set of experiments:

- Rule 1 (S,S,S): $p^1 = -0.97$
- Rule 2 (S,S,L): $p^2 = -0.25$
- Rule 3 (S,L,S): $p^3 = -0.86$
- Rule 4 (S,L,L): $p^4 = -0.39$
- Rule 5 (L,S,S): $p^5 = 0.48$
- Rule 6 (L,S,L): $p^6 = -0.85$
- Rule 7 (L,L,S): $p^7 = 4.03$
- Rule 8 (L,L,L): $p^8 = 0.19$,

where the letter code (e.g., (S,L,S) for p^3) reminds that the corresponding rule was:

IF (x_1 is S_1) and (x_2 is L_2) and (x_3 is S_3) then p^3.

As one can see, the above value function shows that all else being equal, machine value (expected average unit utility expected to be received by the currently running jobs) increases as the first input variable (average unit utility expected to be received by the currently running jobs) increases, the second variable (the expected time remaining until any of the currently running jobs completes its execution) increases, and the third variable (number of free CPUs on the machine) decreases. While this pattern corresponds to our prior expectations, the actual numbers learned with RL allow the scheduler to make specific trade-offs between the relative changes in these variables (e.g., deciding whether a certain increase in the first variable outweighs a certain decrease in the second variable, etc.).

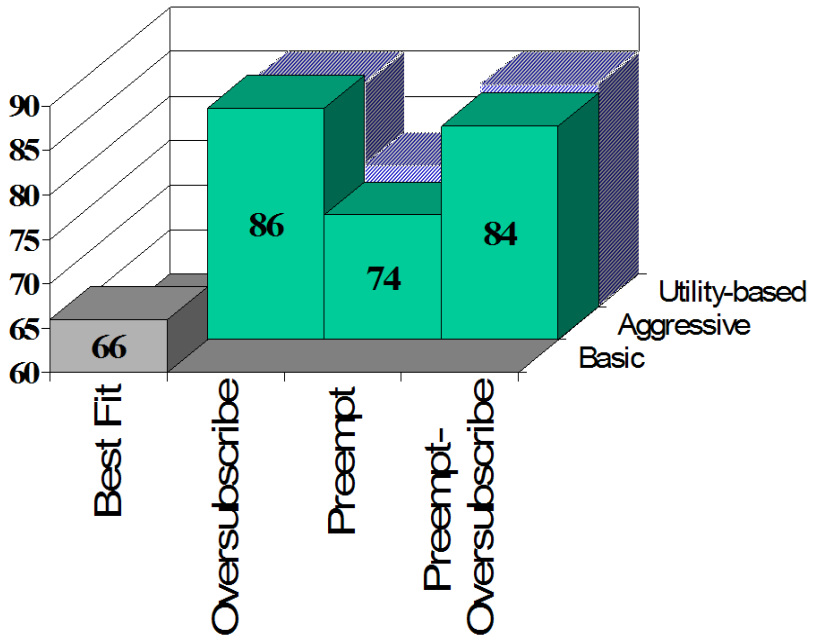

Figure 3. Machine utilization for various policies.

Figure 3 shows average machine utilization (average number of busy CPUs over time) for the policies described above. A comparison of Figures 2 and 3 shows that the policy that maximizes machine utilization (always oversubscribing) does not maximize machine productivity, which is achieved by the utility-based preemption-oversubscribing policy. Our experimental results have also shown that the productivity-optimal policy can achieve machine utilization arbitrarily close to 1 if a high enough job arrival rate is used.

It is also interesting to observe that machine utilization can be computed as:

$$U = \frac{\sum_i n_i t_i}{NT}, \tag{7}$$

where the index i runs over all jobs that were scheduled over time T, n_i is the number of processors used by job i and t_i is its actual execution time, and N is the total number of processors in the system. If the workload is defined as a fixed set of particular jobs (as opposed to a stochastic arrival process), then a higher value of U implies a smaller overall completion time, which does not have to result in a larger machine productivity, as can be demonstrated by the following simple example. Assume that a workload

consists of 4 jobs, each of which can be executed in one time step: a 1-CPU job and a 2-CPU job with decreasing time utility functions (TUFs) as well as a 2-CPU job and a 3-CPU job with constant TUFs. If a 4-CPU machine is available, then co-scheduling together the 2-CPU jobs as well as the 1-CPU and the 3-CPU job maximizes U but does not maximize the total machine productivity, which is maximized by co-scheduling the 1-CPU job and the 2-CPU job with decreasing TUFs in the first place and then scheduling the remaining two jobs one after another.

Machine utilization U as computed in equation (7) in the case of a fixed set of jobs was recently referred to by several researchers as Effective System Performance (ESP) [20], and this term is currently widely used by many supercomputing centers including NERSC. The utility-based productivity measure used in this paper generalizes ESP to the Utility Accrual (UA) paradigm, since it can be expressed as:

$$P = \frac{\sum_i u_i}{NT}, \tag{8}$$

where u_i is the utility obtained by job i upon completion. If job i was scheduled without any wait time and was assigned all the CPUs it wanted, then $u_i = n_i t_i$. Otherwise, $u_i < n_i t_i$ to the extent controlled by the time utility function for job i. The maximum productivity P_{max} that can be obtained from the system in the case of an ideally fitting workload is NT – hence, the denominator does not change. In our case of one machine with 24 CPUs and a 20000 time step testing period, $P_{max} = 480$ thousand utility units, which can be used as the upper limit for numbers in Figure 2.

Finally, sensitivity results were conducted for the utility-based scheduling approach. The benefit of this approach was found to increase as the job arrival rate (system load) increased and more jobs became available for scheduling. Moreover, at high system loads, performance of the benchmark non-adaptive policy started decreasing, since the CPUs were working at full capacity (and completing the same number of jobs per unit of time) but each arriving job waited longer in the queue and hence brought a smaller utility upon completion. The utility-based scheduling policy, however, kept increasing its performance throughout the range of considered arrival rates, as shown in Figure 4.

Furthermore, sensitivity analysis of performance results to the shape of the Time Utility Functions,

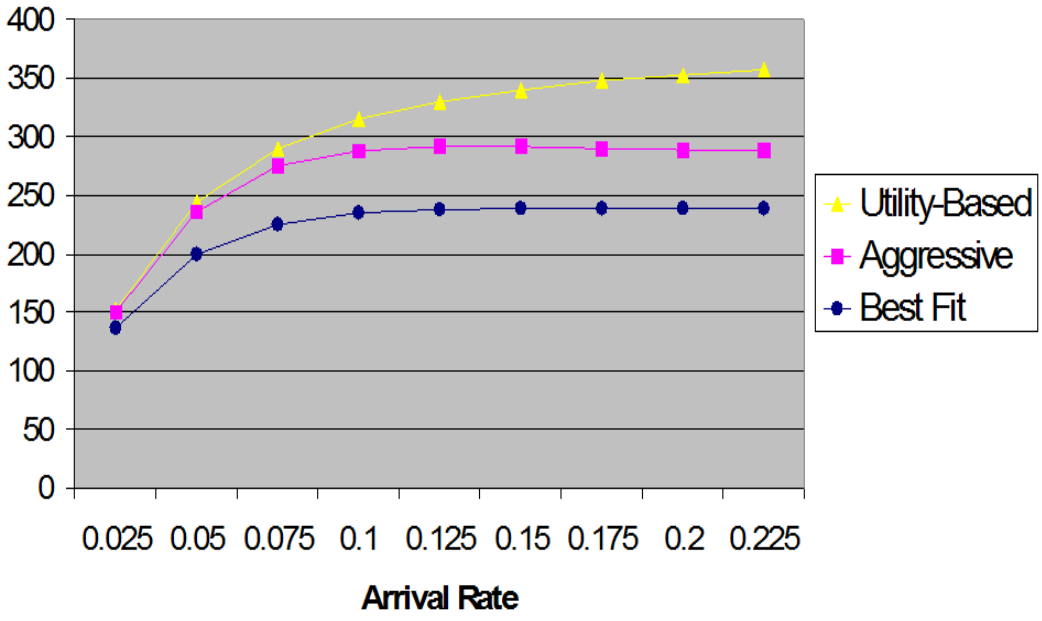

Figure 4. Performance of different scheduling policies as a function of the job arrival rate

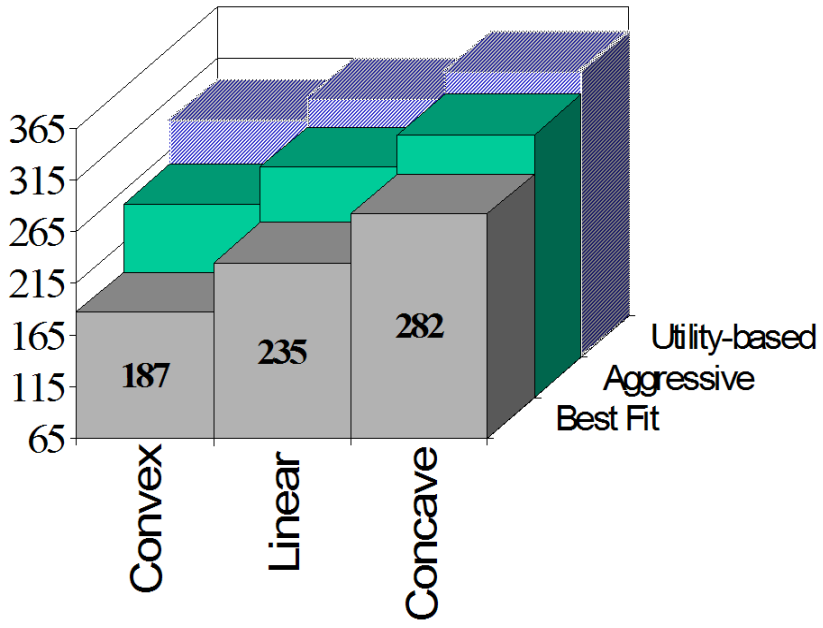

Figure 5. Sensitivity of productivity to the shape of TUFs for job arrival rate of 0.1

presented in Figure 5, showed that performance increases as utility functions become more convex. The linear case in that figure corresponds to the unit utility function shown in Figure 1, the convex case corresponds to f^2 and the concave case corresponds to $f^{1/2}$. The reason for performance increasing with convexity of f is that the urgency for very efficient scheduling increases (since the waiting jobs lose more of their utility per time step by waiting in the queue), and the possible performance gain due to a generally superior utility-based oversubscribing and preemption policies increases.

6 Conclusions

This paper presents a utility-based approach for scheduling jobs in a computing facility and a reinforcement learning algorithm for tuning parameters of the utility (value) functions. Many other details can be added to the basic scheduling framework described in this paper. However, the central idea of scheduling jobs so as to maximize the state "value" of a server or a computing facility is general enough to be used in various scheduling environments: a massively-parallel supercomputer, a data center, a Grid computing facility, etc.

The utility-based policy learning framework allows each machine to adjust its policies independently of other machines. The Time Utility Functions used in this paper not only allow each user to encode its Quality of Service preferences, but also provide a natural optimization objective (total utility of all completed jobs) for any algorithms that perform automatic tuning of scheduling policies. In some environments it might be preferable to ensure that every submitted job gets executed in a fair manner, which can be achieved by *increasing* the utility of each job as it waits in the queue, as opposed to decreasing it.

The presented utility-based scheduling framework can be easily extended to the multi-project domain, where different projects have different priorities. In that case, a job from a higher-priority project would automatically preempt enough of the lower-priority jobs to fit itself (preempting those with lowest remaining utility first), and the utility-based preemption and oversubscribing policies would be invoked only for arbitrating between unscheduled jobs of the same priority.

7 Acknowledgements

The author would like to thank Declan Murphy and Ilya Gluhovsky from Sun Microsystems for helpful comments and corrections to this paper.

References

[1] D. Bertsekas. *Dynamic Programming*. Prentice. 1987.

[2] D. Bertsekas and J. Tsitsiklis, *Neuro-Dynamic Programming*, Athena Scientific, 1996.

[3] R. K. Clark. "Scheduling Dependent real-time Activities," Ph.D. Dissertation, Carnegie Mellon University, CMU-CS-90-155, 1990. http://reports-archive.adm.cs.cmu.edu/anon/1990/CMU-CS-90-155.pdf

[4] E. D. Jensen and J. D. Northcutt. "Alpha: A non-proprietary operating system for large, complex, distributed real-time systems." In Proceedings of The IEEE Workshop on Experimental Distributed Systems, pp. 35-41, Huntsville, AL, 1990.

[5] E.D. Jensen, C.D. Locke, and H. Tokuda. "A time driven scheduling model for real-time operating systems," In Proceedings IEEE Real-Time Systems Symposium, pp. 112-122, 1985. http://citeseer.ist.psu.edu/jensen85timedriven.html

[6] L. P. Kaelbling, L. M. Littman, and A. R. Cassandra. "Planning and Acting in Partially Observable Stochastic Domains." *Artificial Intelligence*, Vol. 101, No. 1-2, pp. 99-134, 1998.

[7] L. P. Kaelbling, L. M. Littman, and A. W. Moore, "Reinforcement learning: a survey." *Journal of Artificial Intelligence Research*, Vol. 4, pp. 237–285, 1996.

[8] J. Kepner, "HPC Productivity: An Overarching View," *International Journal of High Performance Computing Applications: Special Issue on HPC Productivity.* J. Kepner (editor), Vol. 18, no. 4, Winter 2004.

[9] J. Kepner, “HPC Productivity Model Synthesis,” *International Journal of High Performance Computing Applications: Special Issue on HPC Productivity.* J. Kepner (editor), Vol. 18, no. 4, Winter 2004.

[10] P. Li. *Utility Accrual Real-Time Scheduling: Models and Algorithms.* Ph.D. Dissertation, Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University. 2004.

[11] P. Li, B. Ravindran, H. Wu, E. D. Jensen. “A Utility Accrual Scheduling Algorithm for Real-Time Activities With Mutual Exclusion Resource Constraints,” The MITRE Corporation Working Paper, April 2004.

[12] C. L. Liu and J. W. Layland. “Scheduling algorithms for multiprogramming in hard real-time environment,” *Journal of ACM*, Vol. 20, No. 1, pp. 46-61, 1973.

[13] C. Lu, J. A. Stankovic, S. H. Son, Gang Tao. “Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms,” *Journal of Real-Time Systems*, Kluwer Academic Publishing, Vol. 23, No. 1, July 2002 pp. 85-126.

[14] D. Mosse, M. E. Pollack, and Y. Ronen. “Value-Density Algorithms to Handle Transient Overloads in Scheduling,” Proceedings of the 11th Euromicro Conference on Real-Time Systems, June 1999.

[15] M. M. Nassehi and F. A. Tobagi. “Transmission scheduling policies and their implementation in integrated-services highspeed local area networks.” In the Fifth Annual European Fibre Optic Communications and Local Area Networks Exposition, pp. 185-192, Basel, Switzerland, 1987.

[16] The Open Group. *MK7.3a Release Notes*. Cambridge, Massachusetts: The Open Group Research Institute, October 1998.

[17] R.S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[18] J. N. Tsitsiklis and B. Van Roy. “Average Cost Temporal-Difference Learning,” *Automatica*, Vol. 35, No. 11, November 1999, pp. 1799-1808.

[19] Wang, L.-X. “Fuzzy systems are universal approximators,” In Proceedings of the IEEE International Conference on Fuzzy Systems (FUZZ-IEEE ’92), pp. 1163-1169, 1992.

[20] A. T. Wong, L. Oliker, W. T. C. Kramer, T. L. Kaltz and D.H. Bailey, “ESP: A System Utilization Benchmark”, In Proceedings of Supercomputing 2000 (SC2000) conference.

[21] H. Wu, B. Ravindran, E. D. Jensen, and U. Balli. “Utility Accrual Scheduling Under Arbitrary Time/Utility Functions and Multiunit Resource Constraints,” In proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA), Gothenburg, Sweden, August 2004.

8 Vitae

David Vengerov is a staff engineer at Sun Microsystems Laboratories. He is a principal investigator of the Adaptive Optimization project, developing and implementing self-managing and self-optimizing capabilities in computer systems. His primary research interests include Utility and Autonomic Computing, Reinforcement Learning Algorithms, and Multi-Agent Systems. He holds a Ph.D. in Management Science and Engineering from Stanford University, an M.S. in Engineering Economic Systems and Operations Research from Stanford University, an M.S. in Electrical Engineering and Computer Science from MIT and a B.S. in Mathematics from MIT.