1-1-2011

Utility-Aware Scheduling of Stochastic Real-Time Systems

Terry Tidwell
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Thesis Examination Committee:
Christopher Gill, Chair
Roger Chamberlain
Chenyang Lu
Hiro Mukai
Arye Nehorai
William Smart

Utility-Aware Scheduling of Stochastic Real-Time Systems

by

Terry Tidwell

A dissertation presented to the Graduate School of Arts and Sciences
of Washington University in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2011
Saint Louis, Missouri

ABSTRACT OF THE THESIS

Utility-Aware Scheduling of Stochastic Real-Time Systems

by

Terry Tidwell

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2011

Research Advisor: Christopher Gill

Time utility functions offer a reasonably general way to describe the complex timing constraints of real-time and cyber-physical systems. However, utility-aware scheduling policy design is an open research problem. In particular, scheduling policies that optimize expected utility accrual are needed for real-time and cyber-physical domains.

This dissertation addresses the problem of utility-aware scheduling for systems with periodic real-time task sets and stochastic non-preemptive execution intervals. We model these systems as Markov Decision Processes. This model provides an evaluation framework by which different scheduling policies can be compared. By solving the Markov Decision Process we can derive value-optimal scheduling policies for moderate sized problems.

However, the time and memory complexity of computing and storing value-optimal scheduling policies also necessitates the exploration of other more scalable solutions. We consider heuristic schedulers, including a generalization we have developed for the

existing Utility Accrual Packet Scheduling Algorithm. We compare several heuristics under soft and hard real-time conditions, different load conditions, and different classes of time utility functions. Based on these evaluations we present guidelines for which heuristics are best suited to particular scheduling criteria.

Finally, we address the memory complexity of value-optimal scheduling, and examine trade-offs between optimality and memory complexity. We show that it is possible to derive good low complexity scheduling decision functions based on a synthesis of heuristics and reduced-memory approximations of the value-optimal scheduling policy.

Acknowledgments

Completing my PhD has been the most humbling accomplishment of my life. My success is mostly the result of having had the luck and opportunity to work with (and in one case marry) wonderful and incredibly talented people.

My wife Vanessa is the reason you are able to read this today. Her patience and understanding has made all this work possible. My late nights and last minute panics were not mine alone. She has been with me every step of the way, providing insight, reassurance, and occasionally sharp doses of good sense. Her contributions are not limited to emotional support: any math or typographical errors that remain are despite her best efforts to correct them.

My advisor, Chris Gill is the kindest, warmest, most wonderful advisor that I could imagine. He has gone above and beyond with the help and guidance he has given me over the past six years. He has been exceedingly generous with his time, working late nights and weekends. His boundless optimism has been a constant source of reassurance, especially when by necessity the successes of this dissertation were preceded by numerous failures.

I would especially like to thank Rob Glaubius, my collaborator and friend. Our collaboration, of which this dissertation is but the most recent fruit, has been the most rewarding part of my doctoral research experience.

I would also like to thank Bill Smart, Chenyang Lu, and Roger Chamberlain, three professors I've had the honor and pleasure of working with on the various research projects of which I've been a part.

This work was also made possible by a veritable army of talented undergraduates: Carter Bass, Eli Lasker, Micah Wylde, Justin Meden, David Pilla, Braden Sidoti, and Percy Fang. Their contributions were indispensable in bringing this research to fruition.

Finally, my parents. I finally can answer their question: "When will you graduate," even if I may never be able to answer the question "Now, what is it you do again?" to their satisfaction.

Thank you.

Terry Tidwell

Washington University in Saint Louis
August 2011

This research has been supported by National Science grants CNS-0716764 (Cybertrust) and CCF-0448562 (CAREER).

For Vanessa.

Contents

List of Figures

Chapter 1

Introduction

An emerging class of real-time systems, called cyber-physical systems, is characterized by those systems' ability to interact with the physical world through sensing, actuation, or both. These systems are unique among real-time systems in that they have tightly coupled computational and physical semantics. Timing requirements imposed by these interactions are of paramount concern for the safe and correct operation of cyber-physical systems. As with traditional real-time systems, satisfying these timing requirements is the primary concern for the design and development of cyber-physical systems.

In addition to constraints on the system induced by timing requirements, other constraints may also influence the deployment and design of cyber-physical systems. Size, weight, power consumption, or other physical restrictions may limit the resources available, resulting in contention for shared resources. In such cases, a scheduling policy must arbitrate access to shared resources while satisfying system timing and other constraints. The creation of a scheduling policy for such systems is dependent on the abstraction used to represent underlying timing requirements.

1.1 Scheduling Abstractions

Real-time systems traditionally model timing requirements as deadlines. Each schedulable activity, called a *job*, is assigned a deadline. If a job is scheduled so that it completes before its deadline, the deadline is met, and otherwise the deadline is missed.

The deadline scheduling abstraction has several drawbacks. In particular, deadlines have inherent ambiguity. To demonstrate this ambiguity we consider two classes of systems: *hard* and *soft* real-time systems.

The semantics and goal of a scheduling algorithm in real-time systems depend on the exact specifications of the system being scheduled. In *hard real-time systems* [5], any deadline miss may be considered equivalent to a catastrophic failure. The severity of this failure is not specified exactly by the deadline abstraction, but may range from significant performance degradation to total system failure. The goal of the scheduler is to prevent any job from finishing after its deadline. A hard real-time system is not feasibly schedulable unless all jobs can be guaranteed to meet their deadlines.

In contrast, *soft real-time systems* are systems in which a deadline miss is not considered catastrophic. Neither the effects of a deadline miss on the system, nor what happens to the job that missed its deadline are specified under the deadline scheduling abstraction. A missed deadline may simply mean that the opportunity for a job to execute has passed with no additional gain or penalty, or it may mean that some penalty has been incurred by the system, and further penalty may be incurred if the job is not completed as soon as possible. Depending on what the system semantics are, the goal of the scheduler may be, for example, to minimize the number of deadline misses, or to minimize the total lateness of jobs (i.e., the amount of time by which they miss their deadlines).

This lack of precision and expressiveness in capturing timing constraints makes using deadlines as the primary scheduling abstraction for cyber-physical systems unattractive. However despite these drawbacks, deadlines are widely used. First, the theory behind scheduling under the deadline abstraction is very well established, with a variety of scheduling algorithms available. Second, the expressive limitations of deadlines are not a hindrance for many classes of real-time systems in which the closed nature of the design problem allows the desired semantics to be handled implicitly.

Time utility functions (TUFs) are a powerful abstraction for expressing more general timing constraints [20, 41], which characterize the *utility* of completing particular jobs as a function of time and thus capture more complex semantics than simple deadlines. To satisfy temporal constraints in a system whose semantics are described by time utility functions, a scheduling policy must maximize system-wide utility accrual.

These more general and explicit representations of scheduling criteria are compelling for some classes of scheduling problems, and have been studied under particular conditions in other previous research.

However, jobs in cyber-physical systems also have other features that complicate their scheduling problems. Specifically, jobs in many cyber-physical systems may be non-preemptable and may have stochastic duration. For example, jobs may involve actuation, such that preemption of a job may require restoring the state of a physical apparatus such as a robotic arm. In such cases the cost of preemption may be unreasonably high. Therefore scheduling algorithms for certain kinds of cyber-physical systems must be able to consider non-preemptable jobs.

Interaction with the physical environment also may lead to unpredictable job behavior, most notably in terms of job durations. A scheduling policy should anticipate this variability not only by considering the worst case execution time (as is often done in deadline-based scheduling) but also the probabilistic distribution of job durations. This is especially important when the goal is to maximize utility accrual, which depends on the timing of job completion.

Scheduling problems with these concerns may arise in a variety of cyber-physical systems as well as in traditional real-time systems. In mobile robotics, jobs may compete for use of a robotic arm that must be scheduled for efficient alternation. In CPU scheduling, quality of service (QoS) for an application may be specified as a time utility function, and the jobs to be scheduled may have long critical sections where preemption is not allowed. Finally, in critical real-time systems non-preemptive access to a common bus by commercially available off-the-shelf peripherals may have to be scheduled in order to guarantee real-time performance [36].

1.2 Problem Formalization

The problem addressed in this dissertation is the scheduling of a discrete time system in which a set of n *tasks* denoted $(T_i)_{i=1}^n$ are in contention for a shared resource. Each task is composed of an infinite series of jobs, where $J_{i,j}$ refers to the jth job of task T_i. The release time of job $J_{i,j}$ is denoted $r_{i,j}$. After release, the job is added to a

scheduler's *ready queue*. The first job of a task is released at time 0, and subsequent jobs are released at the regular interval p_i, called the period of task T_i. This period allows us to anticipate the arrival of subsequent jobs into the ready queue. A job $J_{i,j}$ with release time $r_{i,j}$ will be followed by a job $J_{i,j+1}$ with release time $r_{i,j+1} = r_{i,j} + p_i$.

Jobs remain in the ready queue until they are scheduled to use the resource, or until they expire due to no longer being able to earn utility. A scheduler for the shared resource repeatedly chooses either to run a job from the ready queue or to idle the resource for a quantum.

When chosen to run, a job is assumed to hold the resource for a stochastic duration. The stochastic duration of each job $J_{i,j}$ is assumed to be an independent and identically distributed random variable drawn from its task T_i's probability mass function D_i. This function is assumed to have support on the range $[1, w_i]$, where w_i is the maximal (commonly referred to as the *worst case*) execution time for any job of T_i and $D_i(t)$ is the probability that a job of T_i runs for exactly t quanta. If job $J_{i,j}$ is completed at time $r_{i,j} + t$, utility is earned as denoted by the task's time utility function $U_i(t)$. This function is assumed to have support on the range $[1, \tau_i]$, where $r_{i,j} + \tau_i$ is the time at which the job expires and is removed from the ready queue. If the job expires before it completes, the system instead is assessed a penalty e_i, where $e_i \leq 0$.

As the system evolves it accumulates utility received from completed jobs and penalties from expiring jobs. The sum of these utilities and penalties is the total utility gain of the system. The goal of a utility-aware scheduler is to maximize this value. Note that this system model is a generalization of both the traditional hard real-time and soft real-time deadline driven model, by setting τ_i to the deadline and e_i to ∞ or 0 respectively.

1.3 Challenges

Optimal utility-accrual and non-preemptive task scheduling problems are both known to be NP-hard [45]. However, these problem features may be common in emerging

cyber-physical systems. Past work in real-time and cyber-physical systems has not adequately addressed the need for schedulers that robustly handle these concerns.

Optimizing utility accrual may be necessary in order to deploy systems that meet the design constraints that these cyber-physical systems are likely to face. This requires an exploration of the trade-off between the cost associated with deriving optimal scheduling policies and the quality of utility-accrual heuristics. In addition, there is a need to examine approaches that trade off optimality for lower memory or time complexity.

1.4 Contributions

This dissertation explores a range of potential solutions to address the need for and the challenges of utility-aware scheduling. Chapter 3 introduces value-optimal utility-aware schedulers. These schedulers can be provided precalculated schedules that maximize expected long term utility. However, these schedulers achieve optimality at exponential cost in memory and time needed to precompute the schedules, which at run-time have low time complexity but high memory complexity. We define a Markov Decision Process (MDP) model of our system that enables us to derive value-optimal schedulers, and also provides a formal framework for comparing the performance of different scheduling policies. We show how our problem structure allow us to bound the number of states in the MDP by wrapping states into a finite number of exemplar states. We give a formal definition of value based on this wrapped MDP which measures expected long term utility density.

Chapter 4 examines heuristics for utility-aware scheduling in comparison to value-optimal schedulers. We adapt existing heuristics for use in the problem domain described in Section 1.2. We adapt the existing Utility Accrual Packet Scheduling Algorithm (UPA) [50] to stochastic real-time systems and arbitrarily shaped timed utility curves. We show under which conditions the examined heuristics perform well compared to value-optimal schedules, and under which conditions heuristic approaches under perform compared to value-optimal schedules. We present a set of recommendations of which heuristics have the best value/cost trade-off for different classes of stochastic real-time systems.

In Chapter 5, we address value-optimal scheduling's high run-time memory complexity. We explore trade-offs between value-optimality and memory cost. We show how heuristics and low-memory approximations of the value-optimal schedule can be synthesized into scheduling decision functions which have low run-time time and memory complexity, but which still achieve a high percentage of the optimal utility that can be gained. We show how decision trees can abstract the structure of the value-optimal policy algorithmically and automatically. This abstraction has reduced run-time memory cost compared to value-optimal. We show that relatively small trees can correctly encode a large percentage of the state-action mappings recommended by value-optimal schedules and have higher value than heuristics. Finally, we show the effects of integrating heuristics into the structure of decision trees.

Chapter 2

Related Work

Time utility functions have been proposed as a method for representing constraints in various systems. A distributed tracking system [8] was proposed for processing data from the Airborne Warning and Control System (AWACS). This tracking system has an upper-bounded number of possible *tracks*, which are streams of sensing data about a particular physical object being tracked. The decisions about what track to process at each point in time are made by the system scheduler. While the system defaults to first-in first-out processing of data, the utility-aware scheduler proposed in [8] allows the system to handle overload scenarios gracefully by prioritizing higher utility tracks.

Time utility functions are also well suited for use in control systems. Control loops may become unstable due to inter-job jitter, the variation in time between job completions [19]. Time utility functions can encode this sensitivity to jitter [17]. By using a utility-aware scheduler these control jobs can be dispatched in such a way as to maximize utility, and therefore minimize inter-job jitter.

Similar quality of service metrics apply in streaming media applications. Inter-frame jitter can cause degradation in playback quality. Because buffering of frames is not always practical, scheduling when frames are decoded may be the better method for controlling playback quality [19].

Utility also has been proposed as a way to schedule communication traffic in Control Area Networks (CAN) in order to guarantee cyber-physical properties such as cruising speeds in automobiles [31].

The concern addressed in this dissertation - scheduling tasks with stochastic non-preemptive execution intervals - is especially relevant in distributed control networks, where it is undesirable to preempt messages already on a CAN and where network delays may be unpredictable. Similar scheduling problems also may occur in real-time systems built from COTS peripherals, where access to the I/O bus may need to be scheduled in order to guarantee real-time performance [36]. Utility-accrual scheduling primarily has been restricted to heuristics based on maximizing instantaneous potential utility density, which is the expected utility of running a job normalized by its expected duration [20]. The Generic Benefit Scheduler (GBS) [24] schedules tasks under resource contention using the potential utility density heuristic without assigning deadlines. If there is no resource contention, the proposed scheduling policy simply greedily schedules jobs according to the highest potential utility density.

Locke's Best Effort Scheduling Algorithm (LBESA) [20,26] schedules jobs with stochastic durations and non-convex time utility functions using a variation of Earliest Deadline First (EDF) [27], where jobs with the lowest potential utility density are dropped from the schedule if the system becomes overloaded. This technique requires an assignment of job deadlines along their time utility curves; optimal selection of those deadlines is itself an open problem.

Other research on utility-accrual scheduling has crucially relied on restricting the shapes of the time utility curves to a single class of functions. The Dependent Activity Scheduling Algorithm (DASA) [9] assumes time utility functions are non-increasing downward step functions. The Utility Accrual Packet Scheduling Algorithm (UPA) [50], which extends an algorithm presented by Chen and Muhlethaler [7], assumes time utility functions can be approximated using a strictly linearly decreasing function. Gravitational task models [17] assume that the shapes of the time utility functions are symmetric and unimodal. In addition it is assumed that utility is gained when the non-preemptable job is scheduled, which is equivalent to assuming deterministic job durations with utility gained on job completion.

No existing utility-accrual scheduling approach anticipates future job arrivals. Therefore, existing techniques are suboptimal for systems in which future arrivals can be accurately predicted, such as those encountered under a periodic task model [27].

MDPs have been used to model sequential decision problems including applications in cyber-physical domains such as helicopter control [34,35] and mobile robotics [23,43]. Our previous work [12, 15, 16] formulated MDPs to design scheduling policies in soft real-time environments with always-available jobs, but did so only for simple utilization-share-based semantics. In this work we extend such use of MDPs to design new classes of *utility-aware* scheduling policies for periodic tasks with stochastic duration.

Several other attempts have been made to address the difficulties that arise from non-preemptive and stochastic tasks in real-time systems. Statistical Rate Monotonic Scheduling (SRMS) [2] extends the classical Rate Monotonic Scheduling (RMS) [27] algorithm to deal with periodic tasks with stochastic duration. Constant Bandwidth Servers (CBS) [6] allow resource reservation in real-time systems where tasks have stochastic duration. Manolache, et al. [29], estimate deadline miss rates for non-preemptive tasks with stochastic duration. These approaches use classical scheduling abstractions such as priority and deadlines, rather than time utility functions, and are thus not appropriate for systems with the more complex timing semantics considered by our approach.

Stochastic models, such as Markov Chains and Markov Decision Processes, have been used in the analysis of schedulers. Examples include calculating the probabilistic response times for interrupt scheduling using Constant Bandwidth Servers [28] and analysis of different Constant Bandwidth Server parameters in mixed hard/soft real-time settings in order to perform distributed scheduling [42]. Analysis of scheduling policies for non-preemptive tasks with stochastic duration [29] has focused on calculating the expectation of a different scheduling metric (deadline miss rates) as opposed to utility accrual. Stochastic analysis also has been applied to global multiprocessor scheduling to calculate expected tardiness for soft real-time systems [32].

Stochastic analysis techniques such as Markov Decision Processes (MDPs) can be used not only to perform analysis, but for design. MDPs are used to model and solve sequential decision problems in cyber-physical domains such as helicopter control [34] and mobile robotics [23, 43].

In previous work we applied MDP based techniques to generating share [12, 15, 16] aware scheduling policies for scheduling tasks with stochastic non-preemptive execution intervals. That work focused on scheduling always-available non-preemptable jobs with stochastic durations to adhere to a desired resource share. This was achieved by penalizing the system in proportion to its deviation from the desired share target. This work extends those techniques to design *utility-aware* rather than *share-aware* scheduling policies, for a periodic task model rather than an always-available job model.

Chapter 3

Markov Decision Process Based Utility-Aware Scheduling

In several important classes of real-time and cyber-physical systems, the ability of a scheduler to maximize utility accrual of non-preemptable, stochastic jobs supports the effective and efficient use of shared resources. However, designing schedulers that maximize utility accrual for such systems is an open research problem. In this chapter we introduce a Markov Decision Process model for the scheduling problem introduced in Section 1.2, and show how techniques from operations research [37] allow us to derive *value optimal* scheduling policies from this model.

3.1 Markov Decision Processes (MDPs)

An MDP is a five-tuple $(\mathcal{X}, \mathcal{A}, P, R, \gamma)$ consisting of a collection of states $\mathcal{X}$ and actions $\mathcal{A}$. The transition system P establishes the conditional probabilities $P(y|x, a)$ of transitioning from state x to y on action a. The reward function R specifies the immediate utility of each action in each state. The reward function R is defined over the domain of state-action-state tuples such that $R(x, a, y)$ is the immediate reward for taking action a in state x and ending up in state y. The discount factor $\gamma \in [0, 1)$ defines how potential future rewards are weighed against immediate rewards when evaluating the impact of taking a given action in a given state.

A policy π for an MDP maps states in $\mathcal{X}$ to actions in $\mathcal{A}$. At each discrete decision epoch k an *agent* (here, a scheduler) observes the state of the MDP x_k, then selects

an action $a_k = \pi(x_k)$. The MDP then transitions to state x_{k+1} with probability $P(x_{k+1}|x_k, a_k)$ and the controller receives reward $r_k = R(x_k, a_k, x_{k+1})$. Better policies are more likely over time to accrue more reward. We then have a preliminary definition of the *value* V^π of the policy π as the expected sum of the infinite series:

$$V^\pi = E\left\{\sum_{k=0}^{\infty} r_k\right\}. \tag{3.1}$$

However, for arbitrary MDPs this sum may diverge, making direct comparison between different policies difficult. To address this issue a discount factor γ is introduced. The value of a policy is thus defined as the expected sum of discounted rewards:

$$V^\pi(x) = E\left\{\sum_{k=0}^{\infty} \gamma^k r_k \middle| x_0 = x, a_k = \pi(x_k)\right\}. \tag{3.2}$$

Overloading notation, we let

$$R(x, a) = \sum_{y \in \mathcal{X}} P(y|x, a) R(x, a, y) \tag{3.3}$$

denote the expected reward when executing a in x. Then we may equivalently define V^π as the solution to the linear system

$$V^\pi(x) = R(x, \pi(x)) + \gamma \sum_{y \in \mathcal{X}} P(y|x, \pi(x)) V^\pi(y) \tag{3.4}$$

for each state x. When $|R(x, a)|$ is bounded for all actions in all states, the discount factor γ prevents V^π from diverging for any choice of policy, and can be interpreted as the prior probability that the system persists from one decision epoch to the next [22]. In practice this value is almost always set very close to 1 (e.g., $\gamma = 0.99$, which was used for the evaluations presented in this dissertation).

There are several algorithms, often based on dynamic programming techniques, for computing the policy that optimizes the value function for an MDP with finite state and action spaces. The optimal value function $V^*(x)$ is defined recursively as:

$$V^*(x) = \max_{a \in \mathcal{A}} \left\{ R(x,a) + \gamma \sum_{y \in \mathcal{X}} P(y|x,a) V^*(y) \right\}$$

Once computed, a corresponding value-optimal policy can be found as defined in the following equation:

$$\pi^*(x) = \operatorname*{argmax}_{a \in \mathcal{A}} \left\{ R(x,a) + \gamma \sum_{y \in \mathcal{X}} P(y|x,a) V^*(y) \right\}$$

Policy iteration [37, 38] converges toward an optimal policy by repeating two steps: policy evaluation and policy improvement.

Policy iteration is initialized with a policy π_0. At iteration k, the value function of π_{k-1}, $V^{\pi_{k-1}}(x)$, is estimated for each state. Based on this estimated value function, the policy is updated during the policy improvement step as follows:

$$\pi^k(x) = \operatorname*{argmax}_{a \in \mathcal{A}} \left\{ R(x,a) + \gamma \sum_{y \in \mathcal{X}} P(y|x,a) V^{\pi_{k-1}}(y) \right\}$$

Because each successive policy is guaranteed to have higher value, and only finitely many possible policies exist, the search is guaranteed to converge [37, 38].

The resulting policy is value-optimal: it optimizes long term value, in contrast to immediate reward. Once computed, this policy can be stored as a lookup table mapping states to actions.

3.1.1 Utility-Aware Task Scheduling MDP

In our system model there are two salient features that determine the *scheduler state*: the system time and the set of jobs available to run. We define our MDP over the set of scheduler states. Each state has two components: a variable t_{system} that tracks the time that has passed since the system began running, and an indicator variable $q_{i,j}$ that tracks whether the job $J_{i,j}$ of task T_i is in the ready queue and can be scheduled.

An action $a_{i,j}$ in our MDP is the decision to dispatch job $J_{i,j}$ and is only valid in a scheduler state if $q_{i,j}$ indicates $J_{i,j}$ is in the ready queue. In addition there is a special action a_{idle}, available in every state, which is the decision to advance time in the system by one quantum without scheduling any job.

However, this basic description of the system state has an infinite number of states, as there are an infinite number of indicator variables needed and the system may run for an unbounded length of time, meaning t_{system} may grow without bound.

In general tasks may have multiple jobs in the ready queue, but because jobs expire, only $\lceil \tau_i/p_i \rceil$ variables are needed to track all the jobs of T_i that can be in the ready queue at one time. In order to limit the number of variables, jobs can be indexed so that the most recently released job of T_i is tracked by the variable $q_{i,1}$. Likewise we index our actions, so that $a_{i,1}$ is the decision to run the most recently released job of T_i. This bounds the number of variables needed to track the state of the ready queue, and consequently bounds the number of different states in which the ready queue can be.

Because t_{system} is not bounded, the resulting MDP still has an infinite number of states. However, the hyperperiod H of the tasks, defined as the least common multiple of the task periods, allows us to wrap the state space of our MDP into a finite set of exemplar states as follows. The intuition is similar to that for hyperperiod analysis in classical real-time scheduling approaches. Given two states x and y with identical $q_{i,j}$ for all tasks but different t_{system} values t_x and t_y such that $t_x \bmod H = t_y \bmod H$, the two states will have the same relative distribution over successor states and rewards. This means that the value-optimal policy will be the same at both states. Thus it suffices to consider only the finite subset of states where $t_{system} < H$. Any

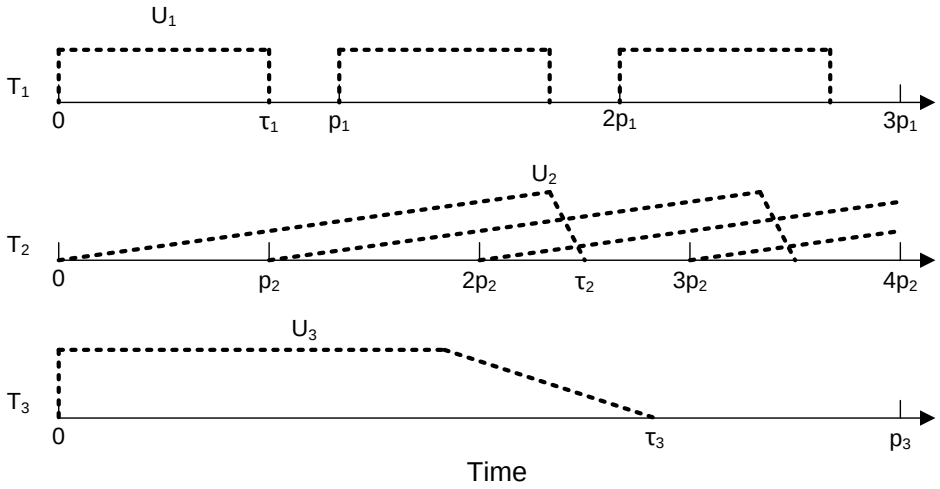


Figure 3.1: Example 3 task system. Periods p_i, time utility function curves U_i, and termination times τ_i are shown for each task.

time the MDP would transition to a state where $t_{system} \geq H$ it instead transitions to the otherwise identical state with system time t_{system} mod H.

If we assume that $\tau_i \leq p_i$, we can simplify our state $q_{i,j}$ down to a single variable q_i defined over $\{0, 1\}$ for each task because no more than one job of a task T_i can be in the ready queue at once. This variable is 1 if there is a job of task T_i in the ready queue and 0 otherwise. Because we require the termination time of the job to be less than or equal to the task period, we need only reason about one job per task at a time. For example, in Figure 3.1 tasks T_1 and T_3 have time utility functions (TUFs) that satisfy this restriction, and so only a single job of each task may be in the ready queue at any time.

If we allow $\tau_i > p_i$ but assume that jobs of the task must be run in order, q_i becomes an integer which counts the number of jobs of T_i that are in the ready queue. The values that q_i can take are bounded by $\lceil \tau_i / p_i \rceil$, the maximum number of jobs of T_i that can be in the ready queue. Note that this is a strict generalization of the case where $\tau_i \leq p_i$. For example, in Figure 3.1 $2p_2 < \tau_2 < 3p_2$. Consequently, in this example at most three jobs of task T_2 may be in the ready queue.

If we allow $\tau_i > p_i$ and also allow jobs of a task to be run out of order, then the full expressive power of our model is needed with variables $q_{i,j}$ that track which of the

$\lceil \tau_i / p_i \rceil$ most recently released jobs of T_i are in the ready queue. Note that this again is a strict generalization of the previous two cases.

An action $a_{i,j}$ in our MDP is the decision to dispatch job $J_{i,j}$ and is only valid in a scheduler state if $q_{i,j}$ indicates $J_{i,j}$ is in the ready queue. If we assume that $\tau_i \leq p_i$ (or that jobs of T_i must be run in order), only a single job of T_i is ever eligible to be run. In this case we can simplify our set of actions to a_i, the action that runs the single eligible job, in addition to the special action a_{idle}.

With this mapping from scheduler states to MDP states in place, the transition function $P(y|x, a)$ is the probability of reaching scheduler state y from x when choosing action a. We discuss the formulation of the reward function for our utility-aware task scheduling MDP next, in Section 3.1.2.

3.1.2 Reward Function

The reward function is defined similarly to the transition function. The scheduler accrues no reward if the resource is idled, i.e., $R(x, a_{idle}, y)$ is always zero. Otherwise, $R(x, a_{i,j}, y)$ is the *utility density* of job $J_{i,j}$ of task T_i that was just run. The utility density of a job $J_{i,j}$ completed at time t is defined as $U_i(t - r_{i,j})/(t - r_{i,j})$. Utility density is used as the immediate reward as opposed to $U_i(t - r_{i,j})$ in order to differentiate between jobs with different durations. It is then possible to define the expected potential utility density and thus the immediate reward for action $a_{i,j}$ in terms of the probability mass function D_i and the time utility function U_i, as a function of the current time t_{system} and the release time of the job $r_{i,j}$ as shown in Equation 3.5.

$$R(x, a_{i,j}) = \sum_{k=1}^{w_i} \frac{D_i(k) U_i(k + t_{system} - r_{i,j})}{d} \tag{3.5}$$

An example calculation of expected potential utility density is shown in Figure 3.2. Task T_i has TUF U_i as shown in the upper graph. D_i is defined on the range $[1, 2]$ with $D_i(1) = 0.5$ and $D_i(2) = 0.5$. The bottom graph shows the expected utility density calculation shown in Equation 3.5 which is defined to be the immediate reward for scheduling the jobs of task T_i at different times after that job's release.

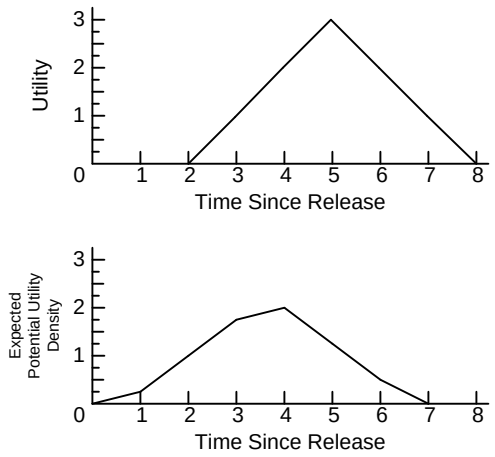


Figure 3.2: Utility function, as a function of completion time, and expected potential utility density as a function of job start time.

Although the equation given in Equation 3.5 works for the general case in which any number of jobs of a task can be run in any order, if we assume $\tau_i < p_i$ we can simplify $R(x, a_{i,j})$ to just $R(x, a_i)$ because only one job of the task will be eligible to run at a time. In this case, the time since the release of the current job of task T_i is then $t_{system} \bmod p_i$. Under this set of assumptions the expected immediate reward $R(x, a_i)$ is:

$$R(x, a_i) = \sum_{d=1}^{w_i} \frac{D_i(d)U_i(d + (t_{system} \bmod p_i))}{d} \tag{3.6}$$

Similarly, if we allow $\tau_i > p_i$ but assume that jobs must be run in order, the time since the release of the job earliest job of task T_i is $p_i(q_i - 1) + (t_{system} \bmod p_i)$ and the immediate reward for taking action a_i is:

$$R(x, a_i) = \sum_{d=1}^{w_i} \frac{D_i(d)U_i(d + p_i(q_i - 1) + (t_{system} \bmod p_i))}{d} \tag{3.7}$$

At time $r_{i,j} + \tau_i$, job $J_{i,j}$ is removed from the ready queue if it has not been run. We account for this by subtracting a cost term $C(x, a_i)$ from the immediate reward function:

$$C(x, a_i) = \sum_{j=1}^{n} \eta_j e_j \tag{3.8}$$

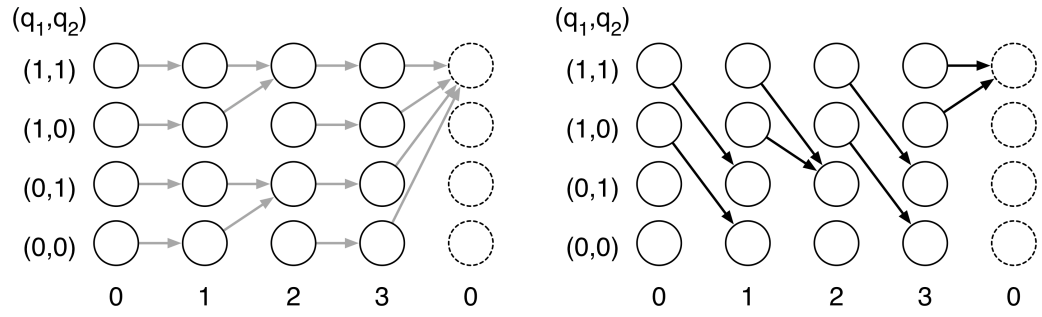
The term n is the total number of tasks, η_i is the expected number of jobs of task T_i that will expire unscheduled given that action a_i is chosen in state x, and the term e_i is the penalty for any job of task T_i expiring. The value of e_i allows us to represent systems with different semantics for expired tasks. For real-time tasks where deadline misses are catastrophic e_i would be set to negative infinity, and any chance of a deadline miss makes this cost term dominate the immediate reward. In contrast, when $e_i = 0$ the only penalty incurred by the system is missing the chance to gain any utility for running the job, and the cost term disappears.

3.1.3 Wrapped Utility-Aware Task Scheduling MDP

A simple but illustrative example of a wrapped utility-aware task scheduling MDP is shown in Figure 3.3. In this example there are two tasks T_1 and T_2 with deterministic quantum durations ($D_1(1) = D_2(1) = 1$), and termination times equal to their periods ($\tau_1 = p_1 = 4$, and $\tau_2 = p_2 = 2$). The states of the MDP are depicted such that states with the same t_{system} are arranged in columns, and states with the same tuple of indicator variables (q_1, q_2) are arranged in rows. Figures 3.3(a), 3.3(c), and 3.3(b) show the transitions between states for taking action a_{idle}, a_1, and a_2, respectively. The states where $t_{system} = 0$ are duplicated on the right side of each figure to illustrate our state wrapping over the hyperperiod (which for this example is 4). Figure 3.3(d) shows the set of reachable states from the state where $q_1 = q_2 = 1$ and $t_{system} = 0$, which is the system's initial state.

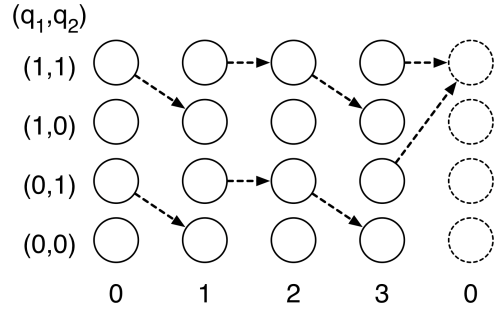
Figure 3.4 shows the effects of non-determinism on the transitions in the MDP. This figures uses the same system parameters as before but now assumes that T_1 has stochastic duration of either 1 or 2 quanta. Only the transitions from the highlighted states are shown.

State wrapping bounds the values that each of the state variables can take, and consequently bounds the size of the state space. If we assume $\tau_i \leq p_i$, an upper

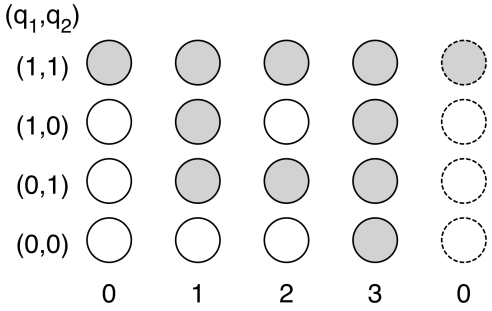


(a) Transitions between MDP states for action a_{idle}

(b) Transitions between MDP states for action a_1



(c) Transitions between MDP states for action a_2



(d) Set of reachable states

Figure 3.3: Wrapped utility-aware task scheduling MDP for a simple but illustrative example of two tasks with periods $p_1 = 4$ and $p_2 = 2$, termination times equal to periods, and (deterministic) single quantum duration. Rows show states with the same valued indicator variables (q_1, q_2), and columns show states with the same t_{system} value.

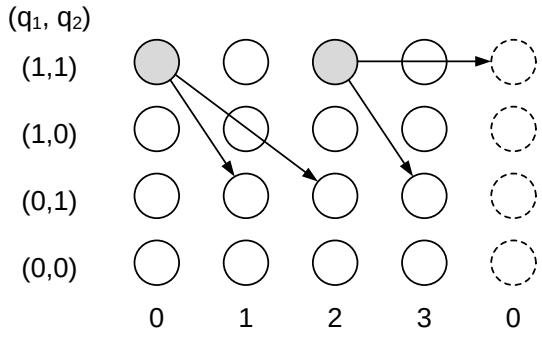


Figure 3.4: Transitions between selected MDP states for stochastic action a_1, where tasks T_1 has stochastic duration in the range $[1, 2]$.

bound for the number of states in the scheduling MDP is

$$H2^n. \tag{3.9}$$

If instead we allow $\tau_i > p_i$ but assume jobs of a task must be run in order, an upper bound for the number of states in the scheduling MDP is

$$H \prod_{i=0}^{n} \lceil \tau_i / p_i \rceil. \tag{3.10}$$

If we allow $\tau_i > p_i$ but let jobs of a task run in arbitrary order, an upper bound for the number of states is

$$H2^{\sum_{i=0}^{n} \lceil \tau_i / p_i \rceil}. \tag{3.11}$$

As stated in the equations above, the size of the state space is sensitive not only to the number of tasks and their constraints but also to the size of the hyperperiod. As such, the solution approach presented in this chapter is especially appropriate for systems with harmonic tasks since in this case $H = \max(p_1, \ldots, p_n)$, which reduces the size of the state space that must be considered. Since real-time systems are often *designed* to have harmonic task periods due to other design criteria (e.g., to maximize achievable utilization under rate-monotonic scheduling of deterministic task sets) it is reasonable to expect this to be fairly common in practice.

Regardless of the assumptions made, the wrapped MDP has finitely many states. Consequently it can be solved using existing techniques from the operations research literature [37]. As was described earlier, the resulting policy for this scheduling MDP optimizes the value function, V^π, defined in Equation 3.2, which is the discounted sum of expected immediate rewards. Because these immediate rewards are defined to be the expected utility density of the scheduling decision, the policy produced optimizes utility accrual, and is value-optimal in the sense that no policy exists that gains more value in expectation. In practical terms this means that the produced scheduling policy is among the co-equal best possible utility-accrual scheduling policies for the modeled system.

The produced policy is stored in a lookup table for use at run-time. The cost of hashing the scheduler state is proportional to its size, resulting in $O(a)$ time overhead,

where a is the number of actions available to the scheduler. For the assumptions in this dissertation this is equivalent to $O(\sum_{i=0}^{n} \lceil \tau_i / p_i \rceil)$, where n is the number of tasks.

3.2 Discussion

By modeling a utility-accrual scheduling problem as a Markov Decision Process we are able to evaluate arbitrary policies' values, and to derive a *value-optimal* scheduling policy. As a policy evaluation method, our Markov Decision Process model has two main advantages. First, it provides a reproducible and rational measure of quality in the face of stochastic behavior. The value calculated by this methodology is the same value that an actual run of the system will have in the limit, over the long term. Second, it accounts for low probability high impact events in calculating the policy value. Unlike quality measures, derived e.g. from Monte Carlo simulation, which may miss a rare event, by taking the expectation from all possible state evolutions, the value of a policy quantifies effects of rare high impact events. This is especially relevant when the penalty associated with a job expiring, e_i, is large compared to the possible utility values.

This formulation as an MDP allows us to derive a *value-optimal* scheduling policy [49] that maximizes the value function for a given discount factor. However, this schedule must be precomputed, and the resulting computation and storage costs can make doing so intractable. To calculate a policy, the full state space of the system must be enumerated and the optimal value function calculated using modified policy iteration [37]. The cost of doing so is polynomial in the size of the state space, which is in turn exponential in the number of tasks. For use at run-time, the value-optimal policy must then be stored in a lookup table, the size of which could be as large as the size of the state space. Because of this, we are interested in how well heuristics with lower computation and storage costs can approximate the value-optimal policy under different system scenarios. In Chapter 4, we describe several relevant heuristics, whose performances are compared to that of the value-optimal policy in an experimental evaluation. In Chapter 5 we introduce reduced-memory approximations of the value-optimal policy in order to address such a policy's run-time memory complexity.

Chapter 4

Utility-Accrual Heuristics

In Chapter 3 we showed how utility-accrual scheduling problems for non-preemptive tasks with stochastic execution times could be represented, under our system model described in Chapter 1, as Markov Decision Processes (MDPs). This allows us to do two things: (1) given a scheduling policy, it allows us to calculate the *value* gained by running that policy, which is a measure of the policy's quality; and (2) given a task set within the system model, it allows us to calculate a *value-optimal* policy, which is a scheduling policy that maximizes the expected value over long term execution.

Sections 4.1 through 4.3 describe relevant scheduling heuristics involving permuting the ready queue (sequencing heuristic), maximizing immediate reward (greedy heuristic), or using deadlines to maximize utility (deadline heuristic). Section 4.4 describes the Utility Accrual Packet Scheduling Algorithm (UPA) [50] and presents new heuristics UPA α and Pseudo α, in which we improve UPA to deal with: (1) stochastic execution intervals, (2) arbitrary time utility function (TUF) shapes, and (3) potential costs and benefits of permuting the ready queue. In Section 4.5 we present an empirical evaluation of these five heuristics across different classes of time utility functions (TUFs), load demands, and penalties for missing deadlines.

4.1 Sequencing Heuristic

A straightforward (if relatively expensive) approach to producing utility-aware schedules is to calculate exhaustively the expected utility gained by scheduling every permutation of the jobs in the ready queue. We define n_q as the number of jobs in the

ready queue, a variable bounded by $\sum_{i=0}^{n} \lceil \tau_i / p_i \rceil$. There are $n_q!$ permutations of the ready queue, and the calculation of expected utility requires convolving the duration distributions of each job in the sequence, an operation that takes time proportional to the maximum task execution time. While impractical for deployment as a scheduler, this heuristic is optimal for a restricted subset of our problem domain in the sense that given no future job arrivals, no work conserving schedule can gain more expected utility. Thus this policy is a good benchmark for measuring what is gained by considering future job arrivals and being non-work conserving (like the value-optimal schedule). We refer to this policy as the *sequencing heuristic*.

4.2 Greedy Heuristic

The *greedy heuristic* is equivalent to the value-optimal schedule that would be generated if the discount factor γ were set to zero. When this happens the scheduler only considers the effect of the expected immediate reward, and not the long term impact of the scheduling decision. Calculating this policy in the soft real-time case, where all penalties for expiring jobs e_i are set to zero, is $O(n_q)$ because the expected immediate reward for each scheduling action can be precomputed and is independent of what other jobs are in the ready queue. In the hard real-time case the immediate reward is not independent of the other jobs, because these other jobs may have a non-zero probability of expiring depending on the stochastic duration of the job under consideration. Calculating this conditional expectation of the expiration cost requires time $O(w)$ where w is the worst case execution among all tasks in the system. Since this must be done for every task, the total complexity is $O(n_q w)$, which may be unacceptably high. Using potential utility density as a heuristic for utility-aware scheduling was proposed in [20, 26]. The Generic Benefit Scheduler [24] uses this heuristic to schedule chains of dependent jobs according to which chain has the highest potential utility density.

4.3 Deadline Heuristic

The Best Effort Scheduling Algorithm [26] uses Earliest Deadline First scheduling [27] as a basis for utility-aware scheduling. Deadlines are assigned to tasks based on the task's time utility function. Although optimal deadline placement is an open problem, deadline assignment is typically done at critical points in the task's time utility function, where there is a discontinuity in the function or in its first derivative. Examples of time utility functions and their critical points are discussed in Section 4.5. We refer to this scheduling algorithm as the *deadline heuristic*.

4.4 UPA α and Pseudo α

The Utility Accrual Packet Scheduling Algorithm (UPA) [50] uses a *pseudoslope* heuristic to order jobs based on the slope of a strictly linearly decreasing approximation of the task's time utility function. This algorithm was developed for use in systems with non-increasing utility functions and deterministic execution times. UPA first selects the set of jobs that will finish before their expiration times, then sorts the rest of the jobs by their pseudoslope (given by $-U_i(0)/\tau_i$). The slope closest to negative infinity is placed first in the calculated schedule. Finally UPA does a bubble sort of the sorted jobs in order to find a locally optimal ordering, in a way similar to the sequencing heuristic discussed in Section 4.1.

To account for time utility functions with arbitrary shapes (as opposed to strictly decreasing time utility functions) our first extension to UPA is to calculate the pseudoslope value using the current value of the utility function at the time when the scheduling decision function is invoked. At time $r_{i,j} + t$, for instance, the pseudoslope value for job $j_{i,j}$ is $-U_i(t)/(\tau_i - t)$.

To make UPA applicable to tasks with stochastic durations, we also introduce a parameter α in $[0, 1]$ that imposes a minimum threshold on the probability that the job will finish before its expiration time. In general *UPA* α considers only jobs whose probability of timely completion is greater than or equal to α, and in particular UPA

0 considers any job while UPA 1 considers only those jobs that are guaranteed to finish before their deadlines.

Finally, with deterministic durations the local search for a better scheduling order is $O(n_q^2)$, but with stochastic durations convolutions of the duration distributions also need to be calculated to find the expected utility of each sequence of jobs. While calculating the expected value of a sequence after an inversion of two jobs in a sequence is $O(1)$ in the deterministic case, in the stochastic case it is $O(w)$. This makes extending this part of the UPA algorithm potentially expensive for online scheduling use. To evaluate the cost and benefit of this sequencing step we define two distinct heuristics: Pseudo α is the scheduling algorithm that simply uses the pseudoslope ordering to schedule jobs, while UPA α performs the additional sequencing step to find a locally optimal schedule.

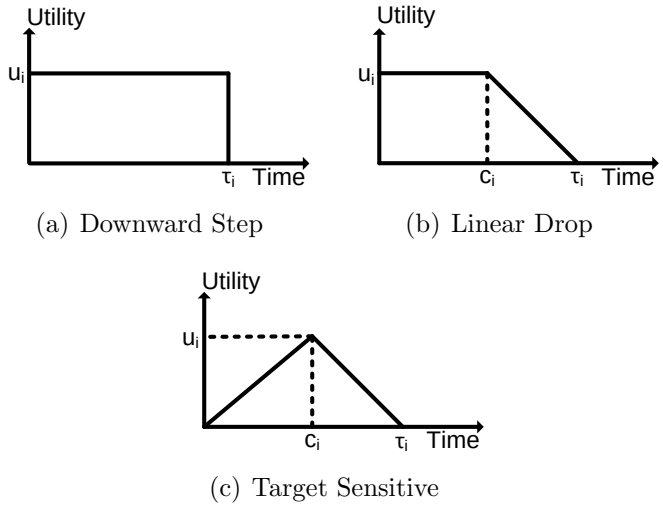
4.5 Evaluation

We first evaluated the heuristics described in this chapter using three different classes of time utility functions which are illustrated in Figure 4.1. A *downward step* utility curve is parameterized by the task's expiration time τ_i and utility upper bound u_i and is defined as:

$$U_i(t) = \begin{cases} u_i & : \quad t < \tau_i \\ 0 & : \quad t \geq \tau_i \end{cases} \tag{4.1}$$

This family of functions is representative of jobs with firm deadlines, like those considered in traditional real-time systems. For the purpose of the deadline heuristic, job deadlines are assigned at the point τ_i. The relative utility upper bounds are important in determining good utility-aware schedules, but are not taken into account by approaches that consider only deadlines.

A *linear drop* utility curve is parameterized like a downward step utility curve, but with an additional parameter describing the function's critical point c_i:



(a) Downward Step

(b) Linear Drop

(c) Target Sensitive

Figure 4.1: Representative time utility functions.

$$U_i(t) = \begin{cases} u_i & : \quad t < c_i \\ u_i - (t - c_i)\frac{u_i}{\tau_i - c_i} & : \quad c_i \leq t < \tau_i \\ 0 & : \quad t \geq \tau_i \end{cases} \tag{4.2}$$

Such a utility function is flat until the critical point, after which it drops linearly to reach zero at the expiration time. This family of curves is representative of tasks with soft real-time constraints, where quality is inversely related to tardiness. For that reason the deadline is set to the point c_i.

A *target sensitive* utility curve is parameterized exactly like a linear drop utility curve, and is defined as:

$$U_i(t) = \begin{cases} t\frac{u_i}{c_i} & : \quad t < c_i \\ u_i - (t - c_i)\frac{u_i}{\tau_i - c_i} & : \quad c_i \leq t < \tau_i \\ 0 & : \quad t \geq \tau_i \end{cases} \tag{4.3}$$

The utility is maximized at the critical point, and is representative of tasks whose execution is sensitive to inter-task jitter. In control systems, whose sensing and actuation tasks are designed to run at particular frequencies, quality is inversely

related to the distance from the critical point. For deadline driven heuristics, this critical point is considered to be the deadline for the purpose of evaluating deadline-driven heuristics.

For the experiments presented in this chapter, τ_i is chosen uniformly at random from the range $(w_i, p_i]$. Because the expiration time precedes the next job's arrival, only one job of a task is available to run at any given time. Because the expiration time is greater than the worst case execution time, the task is guaranteed to complete prior to its expiration time if granted the resource at the instant of release. The upper bound on the utility curve u_i is chosen uniformly at random from the range $[2, 32]$, and the critical point c_i for the target sensitive and linear drop utility curves is chosen uniformly at random from the range $[0, \tau_i]$.

Task periods are randomly generated to be divisors of 2400 in the range $[100, 2400]$, ensuring the hyperperiod of the task set is constrained to be no more than 2400.

The duration distribution for each task is parameterized with three variables (l_i, b_i, w_i) such that $l_i \leq b_i \leq w_i$ where l_i and w_i are the best case and worst case execution times respectively, and that 80% of the probablity mass is in the range $[l_i, b_i]$. The duration distribution for these experiments is defined as:

$$D_i(t) = \begin{cases} 0 & : \quad t < l_i \\ \frac{0.8}{l_i - b_i} & : \quad l_i \leq t \leq b_i \\ \frac{0.2}{b_i - w_i} & : \quad b_i < t \leq w_i \\ 0 & : \quad w_i < t \end{cases} \tag{4.4}$$

This means that the demand of the task, the fraction of the available time that the task requires to complete all its jobs, is normally between l_i/p_i and b_i/p_i but occasionally may be as high as w_i/p_i. The parameters are further constrained such that $l_i/p_i \geq 0.05$ and $b_i/p_i \geq 0.10$.

In addition to the constraints on the individual tasks, constraints are placed on the task set as a whole, such that:

$$\sum_{i=1}^{n} l_i/p_i = L_i$$

$$\sum_{i=1}^{n} b_i/p_i = B_i$$

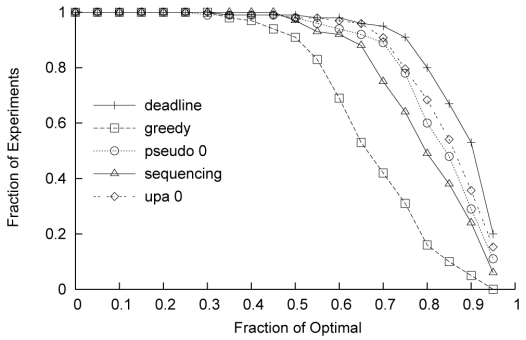
$$\sum_{i=1}^{n} w_i/p_i = W_i$$

where the 3-tuple (L_i, B_i, W_i) defines the overall system load. The default case assumes the values $(0.70, 0.90, 1.20)$, which we call the high load scenario, where the resource is working near capacity with transient overloads. A more conservative case, the medium load scenario, has these values set at $(0.40, 0.51, 0.69)$ which ensures that the system is only loaded up to about 70% capacity, but for the most part is operating at between 40% and 50% capacity. Finally we define a low load scenario which uses the values $(0.07, 0.15, 0.25)$.

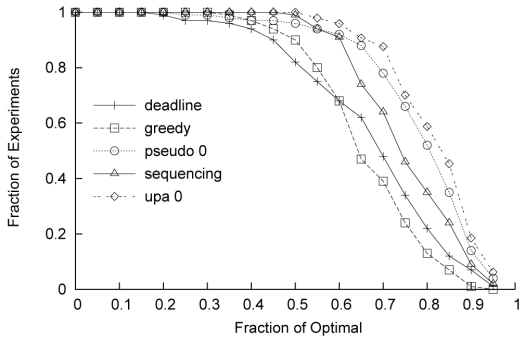
4.5.1 Soft Real-Time Scenarios

In the following experiments, the penalty e_i for a job expiring is assumed to be zero. This means that the heuristics need only consider how to maximize utility accrual, and not how to ensure that certain jobs are scheduled. We begin by focusing on the high load scenario and calculating the value of each heuristic for 100 different 5-task problem instances as a percentage of value-optimal.

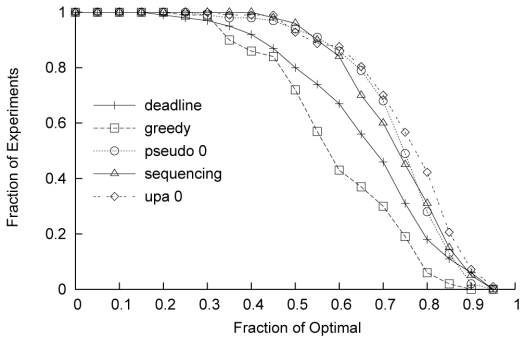
Figure 4.2 shows the results of our experiments. Each graph shows what fraction of the problem instances scheduled by each heuristic achieved at least a given fraction of optimal. Figure 4.2(a) shows that all the heuristics achieved at least 30% of optimal on all problem instances. The greedy heuristic is generally the worst of all the heuristics; less than 20% of the problem instances scheduled using the greedy heuristic achieved at least 80% of optimal.



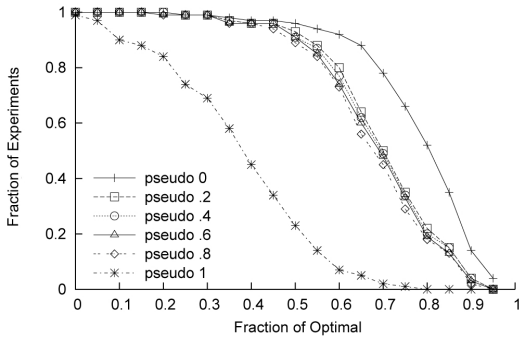
(a) Comparison of heuristics for downward step utility functions.



(b) Comparison of heuristics for linear drop utility functions.



(c) Comparison of heuristics for target sensitive utility functions.



(d) Effect of α on Pseudo α for linear drop utility functions.

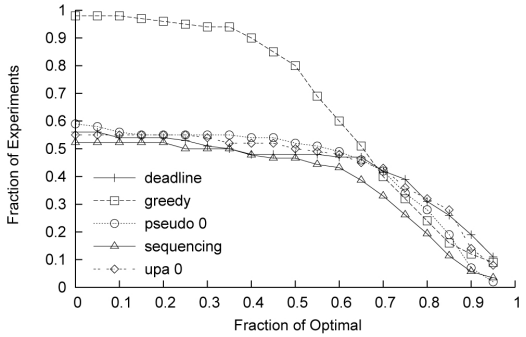
Figure 4.2: Comparison of heuristic policy performance for a soft real-time task set with five tasks under heavy load.

Figure 4.2(a) shows that for soft real-time cases with high load and downward step utility functions, the deadline heuristic performs best. However, as can be seen in Figures 4.2(b) and 4.2(c), the deadline heuristic does not perform as well when the time utility functions are linear drop or target sensitive. Instead UPA 0 performs best, followed closely by Pseudo 0. As was discussed in Section 4.4, this marginal improvement in quality between Pseudo 0 and UPA 0 comes at the cost of a large jump in complexity. This particular value of α was chosen because of the results presented in Figure 4.2(d), which show that by a large margin $\alpha = 0$ outperforms any other setting for this particular case. Although the graph shown is only for linear drop utility functions, the results for downward step and target sensitive were nearly indistinguishable from it. For the soft real-time scenarios we investigated, the value of Pseudo α is maximized by $\alpha = 0$, falls quickly and levels off for values not near 1 or 0, and then deteriorates rapidly again near 1.

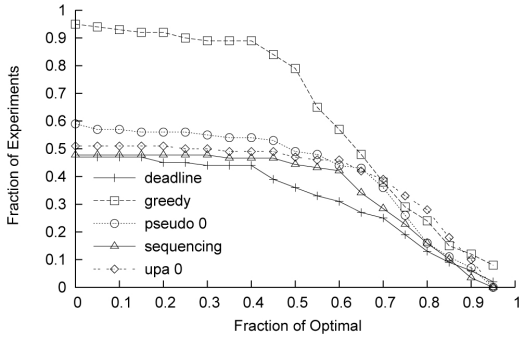
In all soft real-time cases, the greedy heuristic performs relatively poorly, despite greedily maximizing utility density. This is most likely because the utility density is not strictly related to τ_i, and thus a job might have a higher immediate utility density, but still might not be the most urgent job. The sequencing heuristic also performs relatively poorly despite being much more computationally expensive than either UPA 0 or Pseudo 0. This is surprising for two reasons: (1) the scheduling decision made at every point is optimal if we assume that the schedule must be work-conserving and that no more jobs arrive until the ready queue empties; and (2) UPA 0 uses a simplified variation of sequencing to achieve its modest gains over Pseudo 0. Further investigation of this issue remains open as future work.

4.5.2 Hard Real-Time Scenarios

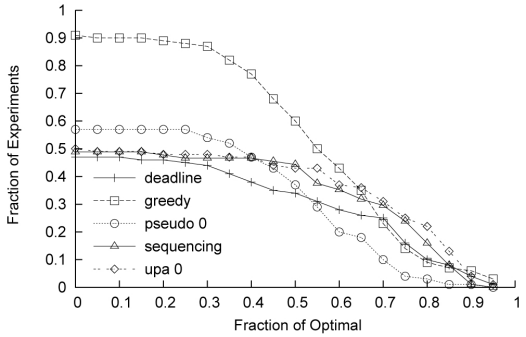
Unlike the soft real-time scenarios presented in Section 4.5.1, in hard real-time systems a task expiring may incur severe penalties. Whereas in the soft real-time case the only risk is the potential loss of utility from not scheduling the job, in the hard real-time case the penalty associated with the job, e_i, may be very large. For the experiments presented in this section one of the five tasks is assumed to be a hard real-time task, with e_i chosen uniformly at random from the range $[-150, -50)$. For the other tasks in the system $e_i = 0$.



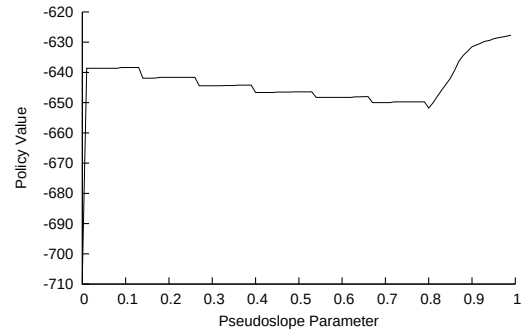
(a) Comparison of heuristics for downward step utility functions.



(b) Comparison of heuristics for linear drop utility functions.



(c) Comparison of heuristics for target sensitive utility functions.



(d) Effect of α on Pseudo α for a single problem instance with target sensitive utility functions.

Figure 4.3: Comparison of heuristic policy performance for a hard real-time task set with five tasks under heavy load.

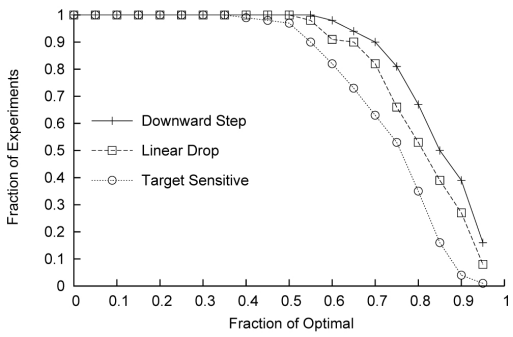
Figure 4.3 shows the results of these experiments. The first difference compared to the soft real-time experiments in Section 4.5.1 is that some of the scheduling policies now achieve overall negative value: that is, they accrue more penalty in the long term than they achieve utility. For instance, Figure 4.3(c) shows that 90% of the target sensitive problem instances scheduled by the greedy heuristic have positive value. Pseudo 0, in contrast, gains positive value in only 60% of the cases. Deadline, sequencing and UPA 0 only have positive value in 50% of the cases.

The differences in performance by time utility function type are also more muted in the hard real-time scenarios. However, Figure 4.3(a) shows that (as before) the deadline heuristic performs best when scheduling problem instances with downward step utility functions, but as Figures 4.3(b) and 4.3(c) show it does not do nearly as well in problem instances with linear drop or target sensitive utility functions.

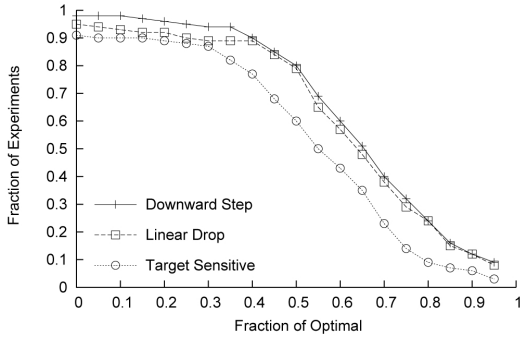
Although the greedy heuristic has the smallest number of problem instances with negative value it is important to remember that, as is mentioned in Section 4.2, calculating greedy is more expensive in the hard real-time case because the immediate reward of a scheduling decision is no longer independent of other jobs in the ready queue. However, it is unclear why the sequencing heuristic, despite similar considerations, degrades sharply. It is possible that because sequencing more efficiently utilizes the resource, hard real-time jobs are more likely to arrive when the resource is occupied and thus expire before completion. Although we report results for Pseudo 0 and UPA 0, Figure 4.3(d) shows that for hard-real time problem instances, it is not clear that there is a single best value for α. In the problem instance shown here, the best value for α is 1, while our other experiments show strong evidence that this is the worst value for α in soft real-time problem instances. It is worth noting that although Pseudo 1 here maximizes expected value, the value is still negative. Further investigation of these issues with the sequencing, Pseudo α, and UPA α heuristics remains open as future work.

4.5.3 Load Scenarios

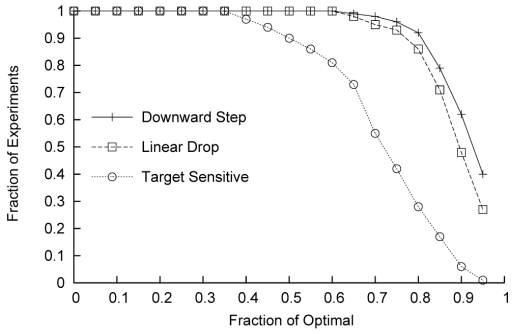
Figure 4.4 shows the effect of different loads on the quality of the scheduling heuristics. Figures 4.4(a), 4.4(c) and 4.4(e) show soft-real time scenarios for different loads



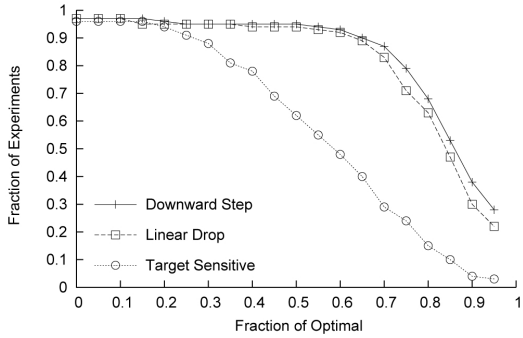
(a) Pseudo 0 in high load soft real-time scenarios.



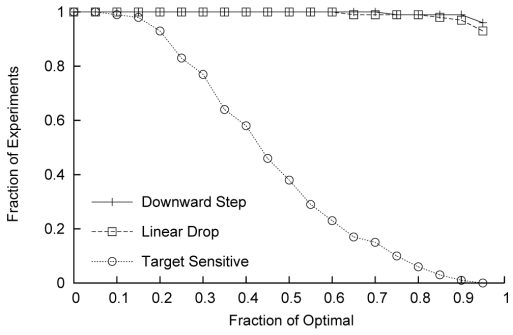
(b) Greedy in high load hard real-time scenarios.



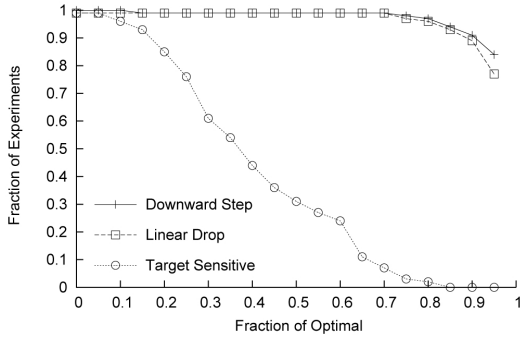
(c) Pseudo 0 in medium load soft real-time scenarios.



(d) Greedy in medium load hard real-time scenarios.



(e) Pseudo 0 in low load soft real-time scenarios.



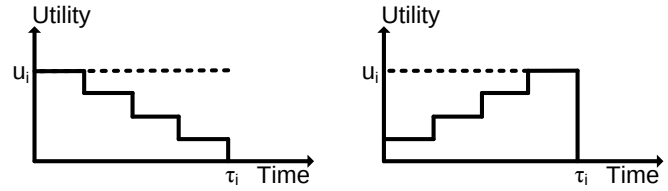
(f) Greedy in low load hard real-time scenarios.

Figure 4.4: Evaluation of selected heuristics for soft and hard real-time cases, for different time utility functions in low, medium and high load scenarios.

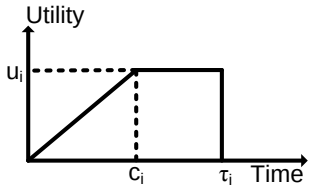
on Pseudo 0, which we found in general offered the best trade-off between value and cost for non-downward step soft real-time scenarios as seen in Figure 4.2. As Figure 4.4(a) shows, Pseudo 0 incurs only minor differences in the quality of the schedules produced in the high load case, regardless of what time utility function class is being scheduled. While Pseudo 0 does slightly better in high load cases when scheduling jobs with downward step utility functions than when scheduling jobs with linear drop utility functions, these differences become even smaller in the medium load scenario shown in Figure 4.4(c). However, even as that happens the value gap between problem instances with these utility functions and the target sensitive utility functions becomes more extreme. This trend continues further in the low load scenario, shown in Figure 4.4(e). In this scenario Pseudo 0 achieves almost identical performance to a value-optimal scheduling algorithm when the time utility functions are either downward step or linear drop. However, problem instances where jobs have target sensitive utility functions are scheduled comparatively poorly.

The poor performance of the Pseudo 0 heuristic in the medium and (especially) low load scenarios may be explained by two factors. First, with less resource contention a value-optimal policy has more degrees of freedom to optimize performance, and therefore heuristic policies like Pseudo 0 achieve a lower percentage of value-optimal. In essence there is more potential utility to be gained, and even a heuristic that achieves the same absolute value in different load scenarios would achieve a lower percentage of the optimal value under low load. Second, work conserving heuristics like Pseudo 0 in low load scenarios are more likely to schedule jobs early when there is little contention for the resource, even though that might result in lower overall expected utility.

Figures 4.4(b), 4.4(d) and 4.4(f) show the effects of load in the hard real-time scenario for the greedy heuristic, which we found to be, in general, the best heuristic overall for hard real-time scenarios as seen in Figure 4.3. The same trends that were seen in the soft real-time case are visible here as well. Once again for all but target sensitive utility functions, greedy did best in the low load scenario, in which almost all problem instances achieved 90% of value-optimal. As in the soft real-time case, low loads made greedy (like the other scheduling heuristics) perform worse on target sensitive tasks, which was most likely caused by similar phenomena to those described in the soft real-time case.



(a) Downward Step Function (b) Upward Step Function



(c) Rise Linear

Figure 4.5: Possible time utility functions.

4.5.4 Other Time Utility Function Effects

As was shown in Sections 4.5.1 and 4.5.2, the shape of the time utility function can have a significant impact on the quality of a scheduling heuristic. Because the shape of a time utility function can be arbitrary (as long as it remains a function mapping from time to utility), an interesting question is whether particular families of curves are particularly difficult to schedule.

To examine this effect we consider additional classes of time utility functions. Unlike the downward step, linear drop, and target sensitive curves, these curves are not inspired by particular tasks in real-time or cyber-physical systems but rather by their potential to reinforce or thwart assumptions pertaining to different heuristics. These curves are shown in Figure 4.5. The first is the *downward step function* utility curve, where utility drops in a series of m flat discrete steps:

$$U_i(t) = \begin{cases} \frac{u_i}{m}\left\lceil \frac{mt}{\tau_i} \right\rceil & : \quad t < \tau_i \\ 0 & : \quad t \geq \tau_i \end{cases} \tag{4.5}$$

The *upward step function* utility curve is similar, but utility rises as the task approaches its expiration time, and then falls off to zero afterward:

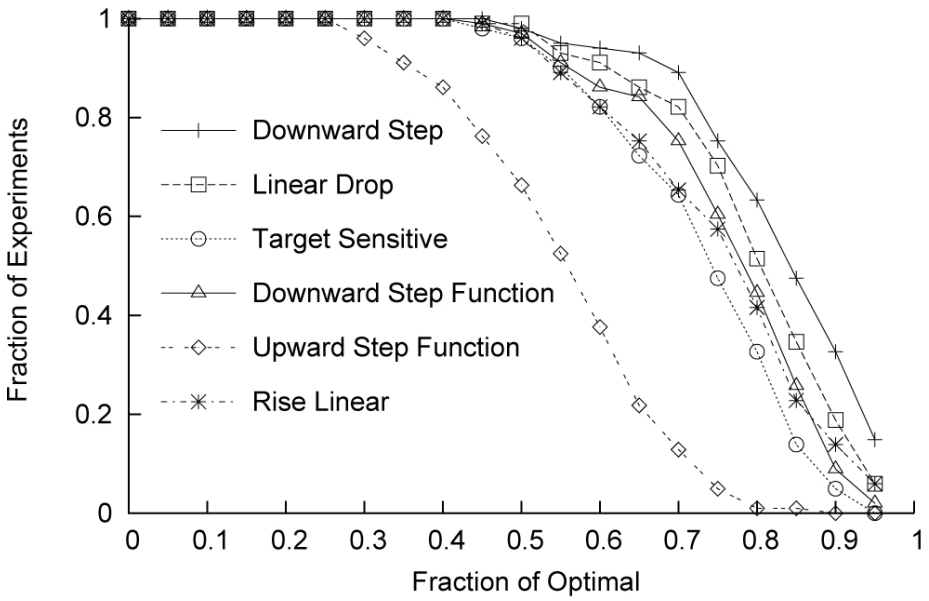


Figure 4.6: Effects of time utility function class on Pseudo 0.

$$U_i(t) = \begin{cases} \frac{u_i}{m}(m - \lceil \frac{mt}{\tau_i} \rceil + 1) & : \quad t < \tau_i \\ 0 & : \quad t \geq \tau_i \end{cases} \tag{4.6}$$

The *rise linear* utility curve is a variation of the target sensitive utility curve, where utility rises up to a critical point c_i and then remains flat, but has an abrupt deadline:

$$U_i(t) = \begin{cases} t\frac{u_i}{c_i} & : \quad t < c_i \\ u_i & : \quad c_i \leq t < \tau_i \\ 0 & : \quad t \geq \tau_i \end{cases} \tag{4.7}$$

To evaluate these effects we ran Pseudo 0 in the soft real-time high load scenario with problem instances created with all six classes of utility curves and 5 tasks. The results of running this experiment on 100 problem instances for each time utility function class are shown in Figure 4.6. Very little differentiates most classes of time utility functions, at least in the heavy load case. The notable exception to this is the upward step function. This likely occurs because Pseudo 0 tries to approximate the

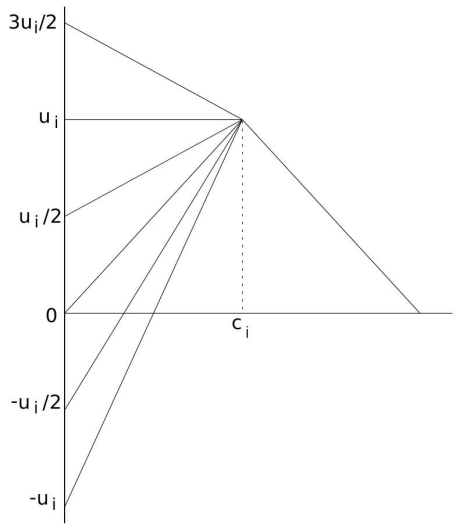


Figure 4.7: Linear drop utility function, with different y-intercepts.

curve at any particular moment as a linearly decreasing function, and the upward step function behaves in a way contrary to this simplified utility model.

A final observation about the classes of time utility functions we consider in this dissertation is that target sensitive and linear drop curves only differ in the slope of the line before the critical point. By changing the slope of the utility curve before the critical point we get the broader class of utility functions shown in Figure 4.7, to which both belong. Curves with y-intercept = 0 are target sensitive utility curves, and curves with y-intercept = u_i are linear drop curves. At any point where $U_i(t) < 0$ we assume instead that $U_i(t) = 0$. The effect of these utility curves on Pseudo 0 is shown in Figure 4.8. The problem instances shown in this experiment are soft real-time 5 task sets with high load. These experiments show that as the utility curve becomes more peaked, Pseudo 0 performs worse compared to value-optimal, which is consistent with the results of our previous experiments.

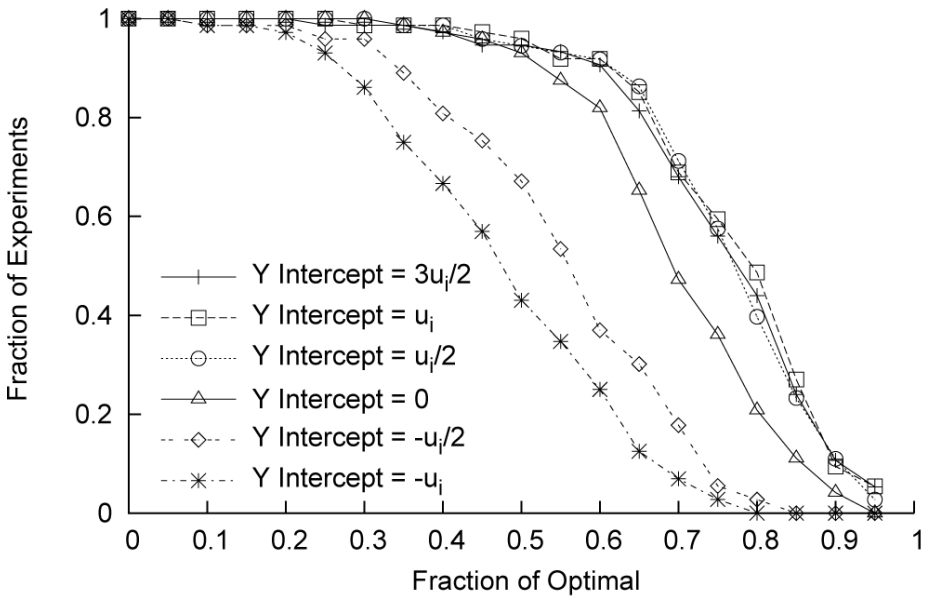


Figure 4.8: Effects of initial time utility function slope on Pseudo 0.

4.6 Discussion and Recommendations

Our MDP formulation allows comparison of various heuristics to a value-optimal policy in order to quantify their performance exactly despite stochastic task behavior. This allows us to determine what percentage of the available utility each of the heuristics achieves under a variety of different scenarios.

From the experimental evidence presented in Section 4.5, the heuristic that provided the best compromise between scheduling complexity and performance for soft real-time systems was Pseudo 0, which extends UPA [50]. There were two notable exceptions, however. First, if the time utility functions of the tasks being scheduled were downward step utility functions, the deadline heuristic was superior to Pseudo 0. Second, in cases with low load and tasks with target sensitive utility functions, in general no heuristic scheduler achieved a significant percentage of the value-optimal objective.

For hard real-time task sets the only acceptable heuristic was the greedy heuristic, albeit with two caveats. First, the value of the policy is not guaranteed to be positive. Second, the scheduling overhead is proportional to the worst case execution time

	High Load	Medium Load	Low Load
Downward Step	Deadline	Deadline	Deadline
Linear Drop	Pseudo 0	Pseudo 0	Pseudo 0
Target Sensitive	Pseudo 0	Pseudo 0	None

(a) Soft Real-Time Guidelines

	High Load	Medium Load	Low Load
Downward Step	Greedy	Greedy	Greedy
Linear Drop	Greedy	Greedy	Greedy
Target Sensitive	Greedy	Greedy	None

(b) Hard Real-Time Guidelines

Figure 4.9: Guidelines for soft and hard real-time scenarios.

among jobs, which could make it computationally too expensive for online use. As in the soft real-time case, in general no heuristic scheduler performed well given low load and tasks with target sensitive utility functions. These guidelines are summarized in Figure 4.9.

Chapter 5

Trade-offs in Value-Optimality Versus Memory Complexity

Chapter 3 introduced a technique for producing value-optimal policies for utility-accrual scheduling problems. However, these policies achieved optimality at the cost of a large up front computation time for precomputing the schedule, and a large run-time memory complexity. Chapter 4 examined heuristic schedulers with low time and memory complexity at run-time, but which achieve a varying fraction of the value of the value-optimal policy. In order to address the needs of utility-accrual schedulers in real-time embedded and cyber-physical systems, in this chapter we consider how to synthesize these approaches to maximize policy value, while maintaining low run-time memory and time complexity.

As was seen in Chapter 4, for some scheduling problems solving the scheduling MDP produced scheduling policies with significantly higher values than any of the heuristics we evaluated. For some scenarios, it thus may be preferable to pay the initial cost for precomputing the value-optimal schedule, and store the resulting policy in a lookup table in which each state maps to the value-optimal action to take in that state. However, this approach suffers from two potential pitfalls: (1) memory complexity, and (2) brittleness due to a lack of generalization, i.e., an inability for the value-optimal policy to adapt if the system visits a state not predicted by the model, which thus has no entry in the lookup table.

The memory requirements for the stored lookup table representation of the value-optimal scheduling policy may be potentially quite large, and thus may require storage in main memory. The modestly sized scheduling problems shown in Section 4.5 may

have state spaces that run into the tens of thousands of states, and each additional task may double the size of the state space. Even if large amounts of time and memory are available to precompute the value-optimal schedule, e.g., off-line during design time, the run-time constraints may still prove prohibitive, especially on memory-limited embedded systems. In order to be usable in such domains, an MDP based scheduling policy thus must address the exponential memory costs of the lookup table.

Because the MDP based scheduling policy is solved over an ideal model of the system it also may not capture the full range of activities that can occur in a real system. Concretely, a state encountered in the real system may not be reachable under the assumptions of the model. This can occur, for instance, if the actual duration distribution of a task is different than that used in the MDP model, e.g., if the actual task has a non-zero probability of running for a duration greater than the worst case execution time captured in the model. In such cases a look-up in the stored policy may fail, resulting in unspecified scheduler behavior. Thus, additional flexibility of the scheduling policy to handle such cases is desirable, and a mechanism is therefore needed to generalize the policy to handle unexpected states.

This chapter describes how both of these issues can be addressed by leveraging decision trees [39,40] to transform the tabular representation of the value-optimal scheduling policy into a functional form. In Section 5.1 we introduce decision trees and algorithms for building them from scheduling policies, and in Section 5.2 we consider the implications of the approach for choosing ideal tree sizes. In Section 5.3 we then evaluate the impact of this approach for representative scheduling problem instances. In Section 5.4 we introduce extensions to the decision tree representation presented in Section 5.1 and evaluate the effects of these extensions on representative scheduling problem instances.

5.1 Decision Tree Representation of Scheduling Policies

Decision trees compactly encode a policy from a collection of labeled instances, which in the problem domain we consider is the set of scheduling states $\mathcal{X}$. Every

$x \in \mathcal{X}$ has a vector of k input variables $x_1, x_2, ...x_k$. The set of possible labels for each x is in $\mathcal{A}$, the set of actions that can be chosen in a scheduling state. A decision tree maps each $x \in \mathcal{X}$ to an $a \in \mathcal{A}$.

Given an unlabeled query state x_{query}, we can then find the label recommended by the decision tree. Every non-leaf node in the decision tree is decorated by a predicate over the vector of k input variables in x_{query}. A predicate P maps x_{query} to 0 if the predicate is false given the values of input variables in x_{query} or 1 if the predicate is true over the input variables. Each non-leaf node has two children with one edge labeled 0 and the other labeled 1. Each leaf node is decorated with a label $a \in \mathcal{A}$.

The set of input variables to the predicate is derived from the state variables of the scheduling state. The first input variable is the system time t_{system}. This variable is bounded in the range $[0, H)$ where H is the hyperperiod of the task set. The next input variable is an indicator variable for each possible job, specifying whether or not it is in the ready queue. This variable takes a value of either 0 or 1. We then introduce an input variable per task which counts the number of jobs of each task in the ready queue. This variable is bounded in the range $[0, \lceil \tau_i/p_i \rceil]$ for task T_i. Finally we introduce an input variable tracking the total number of jobs in the ready queue. This variable is bounded in the range $[0, \sum_{i=0}^{n} \lceil \tau_i/p_i \rceil]$ where n is the number of tasks in the system. The total number of input variables, all of which are integer-valued, is $2 + n + \sum_{i=0}^{n} \lceil \tau_i/p_i \rceil$, where n is the number of tasks.

Given a tree and a query state, a simple recursive algorithm returns a label from the decision tree. Starting with the root node, if the current node of the tree is a leaf node, the algorithm returns the label of the node and terminates. If it is a non-leaf node, the algorithm resolves the predicate labeling the node on x_{query}. If the predicate resolves to false, the algorithm recurses on the child node reached by the edge labeled 0. Similarly, if the predicate is true, the algorithm recurses on the child node reached by the edge labeled 1. In this way, the algorithm searches down the tree for the first leaf node it encounters, and returns the label of that node.

Given a set of states and actions, we wish to build such a decision tree. We consider an algorithm based on the ID3 and C4.5 algorithms [39, 40]. The input to the algorithm is a set $\mathcal{I}$ of labeled instances. An element in $\mathcal{I}$ is a pair in the form (x, a) where $x \in \mathcal{X}$ and $a \in \mathcal{A}$. In the context of our scheduling problem x is a scheduling state

and a is the scheduling action $\pi^*(x)$, which is recommended by the value-optimal scheduling policy.

Given a finite set of predicates, these algorithms recursively build the decision tree. For each predicate P the current set of labeled instances $\mathcal{I}$ is partitioned. The set we consider contains all predicates of the form $x_k < y$, where x_k is an input variable for scheduling state x, and y is an integer value in the range of that input variable. The range for each variable in the vector of input variables is bounded, so there are only finitely many predicates of this form.

The predicate partitions this original set into the two following subsets:

$$\mathcal{I}_{P(x)=0} = \{(x, a) | (x, a) \in \mathcal{I}, P(x) = 0\} \tag{5.1}$$

and

$$\mathcal{I}_{P(x)=1} = \{(x, a) | (x, a) \in \mathcal{I}, P(x) = 1\}. \tag{5.2}$$

The ID3 and C4.5 algorithms greedily build decision trees by maximizing the *information gain* at each level of the tree. This metric attempts to minimize the size of the tree, while at each level of the tree maximizing the accuracy of the encoding of the policy.

The information gain resulting from splitting the original set $\mathcal{I}$ on the predicate P is defined in terms of the *entropy* of the resulting sets. The entropy of a set is defined in terms of the following subset:

$$\mathcal{I}_i = \{(x, a) | (x, a) \in \mathcal{I}, a = i\} \tag{5.3}$$

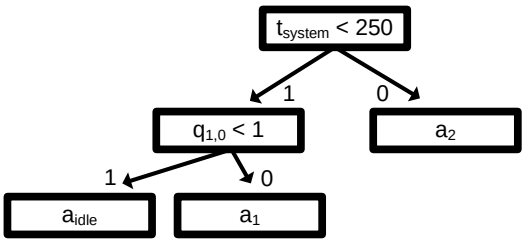
which is the subset of the set $\mathcal{I}$ with the label i. The entropy $h(\mathcal{I})$ of a set $\mathcal{I}$ is in turn a measure of the homogeneity of the labels in the set and is defined in terms of the cardinality of the set $|\mathcal{I}|$, and of each subset $|\mathcal{I}_i|$:

$$h(\mathcal{I}) = -\sum_{a \in \mathcal{A}} \frac{|\mathcal{I}_a|}{|\mathcal{I}|} \log_b \frac{|\mathcal{I}_a|}{|\mathcal{I}|} \tag{5.4}$$

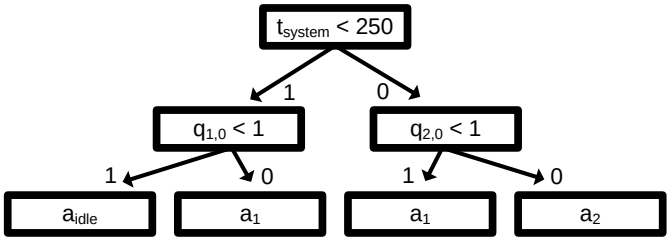
The information gain of a predicate $P(x)$ is calculated as:

$$I_G = h(\mathcal{I}) - h(\mathcal{I}_{P(x)=0}) - h(\mathcal{I}_{P(x)=1}) \tag{5.5}$$

The predicate with maximal information gain separates the original set into two subsets with the greatest homogeneity of labels. If the original set already only has one label, the decision tree building algorithm creates a node labeled with the appropriate label a and the algorithm terminates. Otherwise the predicate that maximizes information gain is used to label the current node of the decision tree. The sets $\mathcal{I}_{P(x)=0}$ and $\mathcal{I}_{P(x)=1}$ are then used as the input sets for recursively labeling the child nodes. Example decision trees illustrating such an expansion are shown in Figure 5.1.



(a) Example decision tree.



(b) Decision tree after expansion of rightmost leaf node.

Figure 5.1: Example decision trees, with predicates defined over state variables.

Because the decision tree may need to grow to exponential size in order to encode an arbitrarily complex policy fully, the algorithm can be configured to terminate either after the policy is fully encoded, or after a given number of splits is reached. At

each iteration, if neither of these termination conditions is reached, the unexpanded node with the highest potential information gain is split into two child nodes. If the algorithm reaches its maximum number of splits, unsplit leaf nodes may be labeled with the most common action of their input set.

Given a decision tree encoding a scheduling policy, we can translate that tree (trivially, using nested conditional logic) into a function that takes as input a scheduler state, transforms it into a vector as described above and follows the decision tree logic until an action is returned. In the worst case, to encode the value-optimal policy fully, a decision tree may need a number of leaf nodes equal to the number of states in the lookup table for that policy. To produce a reduced memory approximation we build the decision tree only to a given depth. In general, this usually results in at least some errors encoding the actions of the value-optimal policy. This may also result in actions recommended by a partially built decision tree being unavailable in the input system state. To address this complication, if the action encoded by the tree is not possible in the current state, the scheduling decision function can return a_{idle}, the action that simply idles the resource.

The run-time complexity of the resulting scheduling decision function depends on the size (i.e., the number of splits) of the tree, d. If we assume that the resulting decision tree is balanced, the worst case time complexity at run-time is $O(\log_2 d)$. The maximal worst case time complexity occurs if the resulting tree is maximally unbalanced in which case traversal requires at most $O(d)$ steps. The memory requirement of the scheduling decision function is $O(d)$. In the experiments described in Section 5.3 we bound the potential size of the decision tree to be less than one hundred nodes, so $d << |\mathcal{X}|$, the size of the scheduling state space. Thus the run-time memory complexity of the tree-based approximation is expected to be much lower than the run-time memory complexity of the lookup table for the value-optimal policy, which is $O(|\mathcal{X}|)$.

5.2 Effects of Decision Tree Representation

The tree building algorithm described in Section 5.1 uses entropy minimization to generate decision trees efficiently. This maximizes the information gain with each split

in the tree. However, it is not information gain itself that we are primarily interested in maximizing. Rather, we are interested in decision trees that (1) accurately encode the state to action mapping of the value-optimal scheduling policy with reduced memory cost and (2) accrue high value. Maximizing the information gain at every iteration of the tree building algorithm is not strictly guaranteed to increase either of these relevant metrics. A simple example can illustrate this phenomenon. Consider the following subset of optimal actions for a set of states:

$$\{a_i, a_i, a_{idle}, a_{idle}, a_i\} \tag{5.6}$$

The most common action in this subset is a_i. If a leaf node of a decision tree encoded the mapping of a set of states to these actions, the leaf node would be labeled a_i by the tree building algorithm. Therefore, for any of the states in the set the policy action encoded by the decision tree would be a_i. If, however, the action a_i were not available in a given state, the scheduling decision function based on this decision tree would instead return a_{idle}, so that no illegal actions are returned by the scheduler. In that case, if the two instances of a_{idle} in the above set are associated with states where a_i is not a legal action, the error rate for the states encoded by this leaf node would be 0.

However, because the actions are not homogeneous, the tree building algorithm instead might split this leaf node. It is possible for a predicate to split the above set into the following two subsets:

$$\{a_i, a_i\}, \{a_{idle}, a_{idle}, a_i\} \tag{5.7}$$

which have a lower combined entropy. Note that the dominant action in the second subset is now a_{idle}. Using the most common label for each set will now introduce an error in the scheduling decision function, whereas the encoding from the smaller tree perfectly encoded the optimal policy for the subset of system states.

This serves to demonstrate that as the size of the tree increases, the value of the approximation produced, at least as measured by the accuracy of the state-to-action

mapping, does not necessarily increase. This means we should have no expectation that a less direct metric such as the value of the resulting scheduling decision function, should monotonically increase as a function of tree size. Therefore, it is necessary to evaluate such effects empirically as is discussed in Section 5.3.

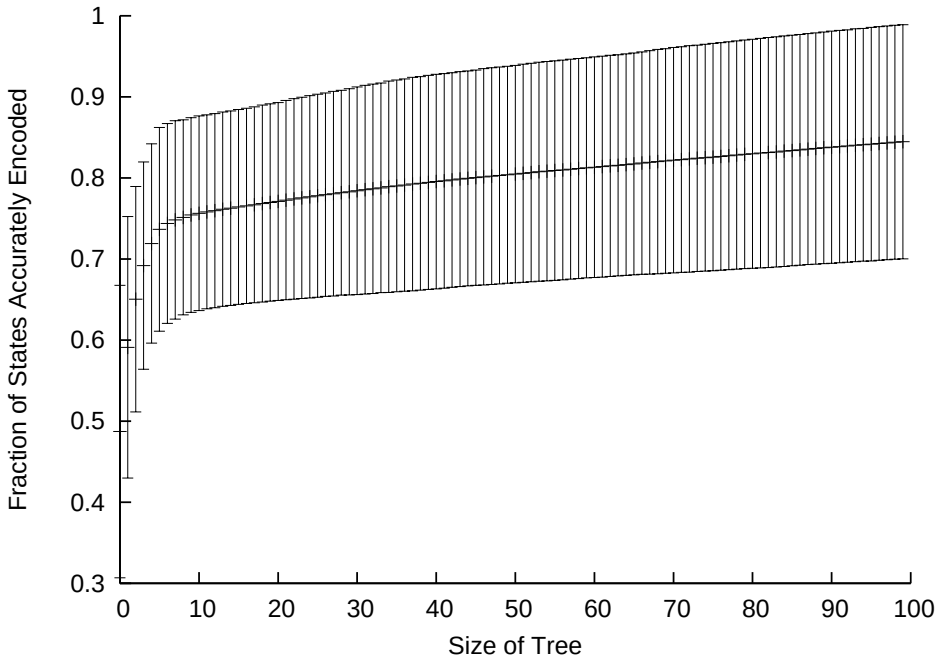
5.3 Experimental Results

In this section we explore the quality of the reduced memory approximations to the value-optimal policy. We examine how the accuracy of the encoded policy varies with the size of the underlying decision tree representation. We also examine the value of the decision tree based scheduling decision function. Finally, given a set of heuristics and different reduced memory approximations of value-optimal policies, we examine how the best of the presented scheduling techniques can be chosen to improve feasible overall system utility accrual, and how the decision tree approach performs in comparison.

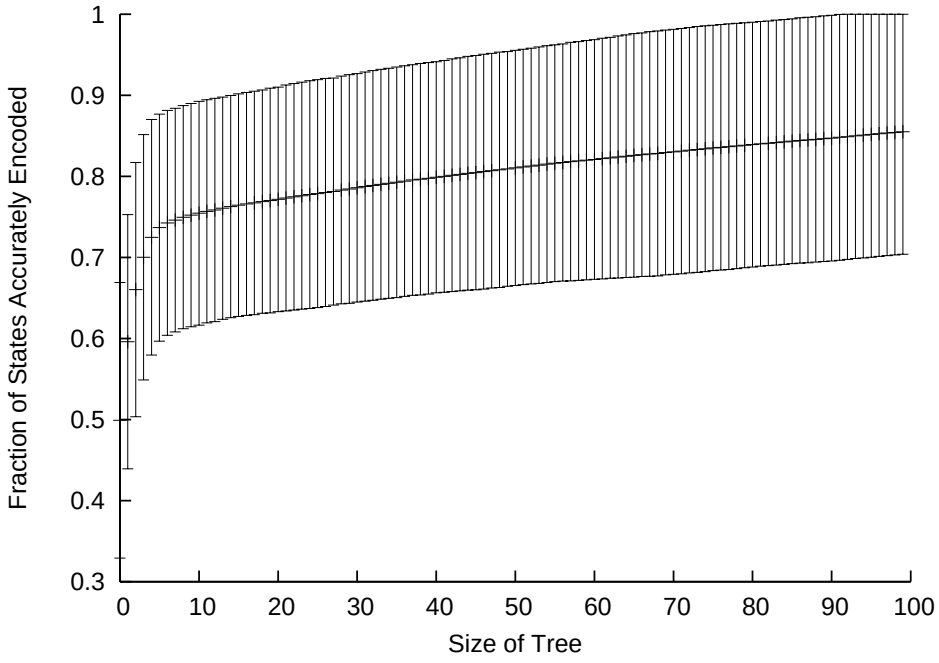
5.3.1 Variation in Accuracy of Encoding with Tree Size

We first examine how accurately decision trees of different sizes can encode the state-action mapping of a value-optimal policy. Specifically, we measure what percentage of the states in the system model are correctly labeled with the value-optimal scheduling action by approximations with different tree sizes.

We generate task sets for the following experiments using the methodology described in Section 4.5. For both hard and soft real-time scenarios, 300 5-task scheduling problem instances were generated: 100 each for the downward step, linear drop and target sensitive time utility function types. All scheduling instances assumed high load, which as is shown in Chapter 4 is a scenario in which heuristics typically have a lower relative value compared to the value-optimal policy. The value-optimal policy for each task set was calculated and then compressed into scheduling decision trees of varying sizes.



(a) Compression of 5-task soft real-time policies.



(b) Compression of 5-task hard real-time policies.

Figure 5.2: Accuracy of policy encoding as a function of the size of the tree, counted as the number of splits. Average accuracy with 95% confidence intervals, based on all 300 problem instances, is shown.

Figure 5.2 summarizes the results of this experiment, where the size of the decision tree is again defined as the number of splits. For this experiment we looked at trees with between 0 and 100 splits. As can be seen, there was no discernible difference between the overall accuracy of the decision tree encoding under the hard and soft real-time cases. We also found no discernible difference in the case of different time utility function types. In general, the larger the scheduling decision tree, the higher the accuracy of the scheduling decision function. However, this trend appears to be logarithmic, suggesting that smaller trees will quickly capture the structure of the policy, while larger trees will only gradually refine this approximation, if at all.

Although the general trend shows improvement in the accuracy of the encoding of a policy's state-to-action mapping, a close examination of the data showed that the effect described in Section 5.2 did in fact occur, but that the number of states affected was generally a very small percentage of the size of the state space.

5.3.2 Variation in Value with Tree Size

Figure 5.3 shows how the value of the resulting reduced memory approximation can fluctuate as a function of tree size, for a randomly selected problem instance. This particular problem is a 5-task soft real-time scheduling problem with a time utility function from the linear drop family of curves. The values of two heuristic policies are shown for comparative purposes. This example illustrates how the value of the reduced memory approximation may be expected to vary for trees whose size is much less than the size of the scheduling problem's state space. Because such variations tend to stabilize above a size of several tens of splits, we evaluate the values of all the reduced value-approximations based on relatively small trees, i.e., trees with between 0 and 100 splits.

It is also appropriate to examine the sizes of the trees that result in the scheduling policies with maximum value. The size of each tree that produced the approximation with the maximum value for our experimental scheduling problems is shown in Figure 5.4. As can be seen in Figure 5.4, a non-trivial number of very small trees (less than 10 splits), produce the reduced memory approximation with the highest value.

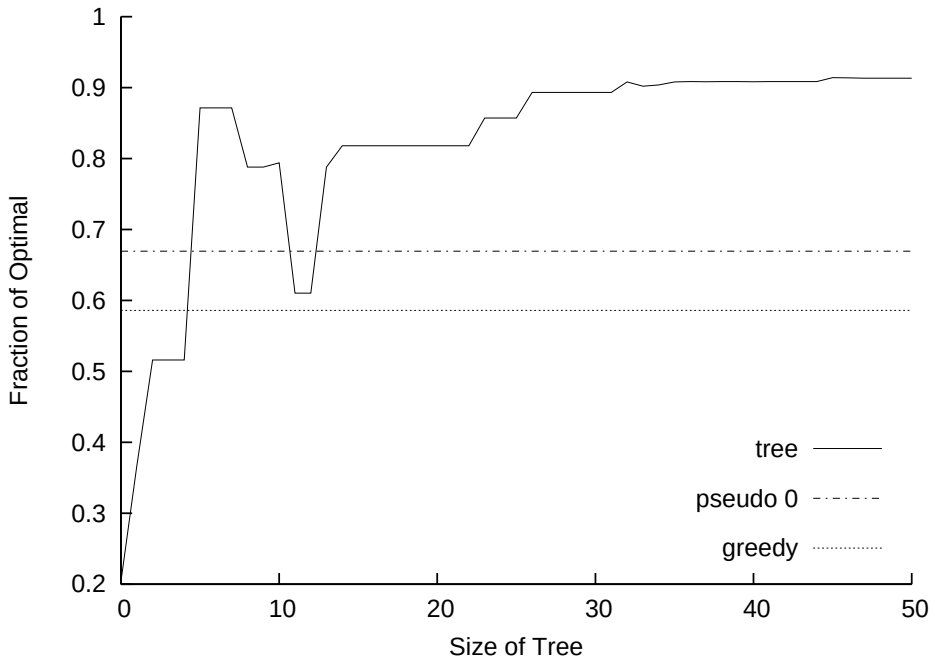
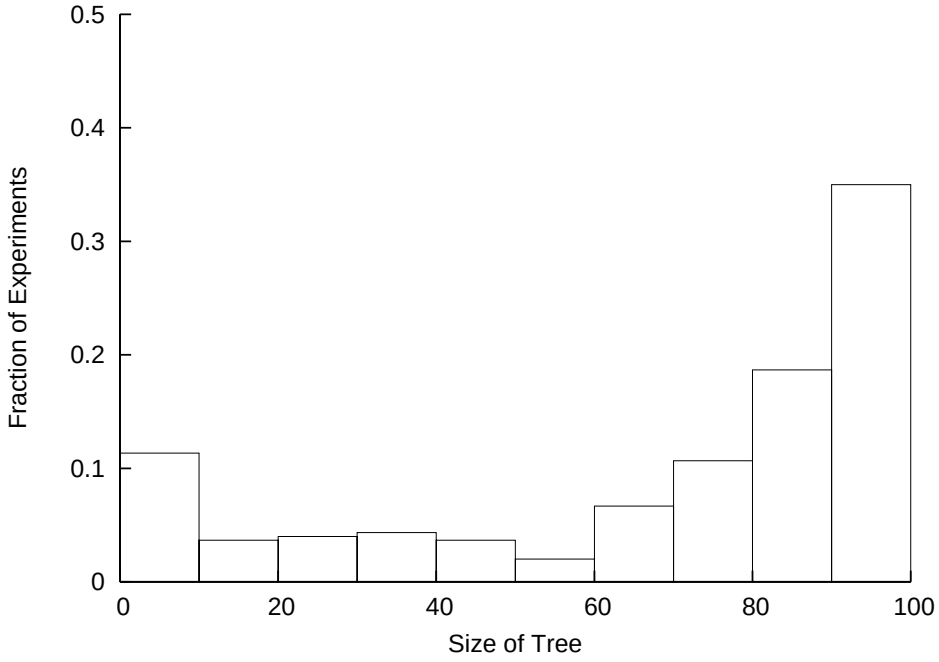


Figure 5.3: Value of tree-based scheduling policy as a function of tree size, compared to the value of the heuristic policies Pseudo 0 and greedy.

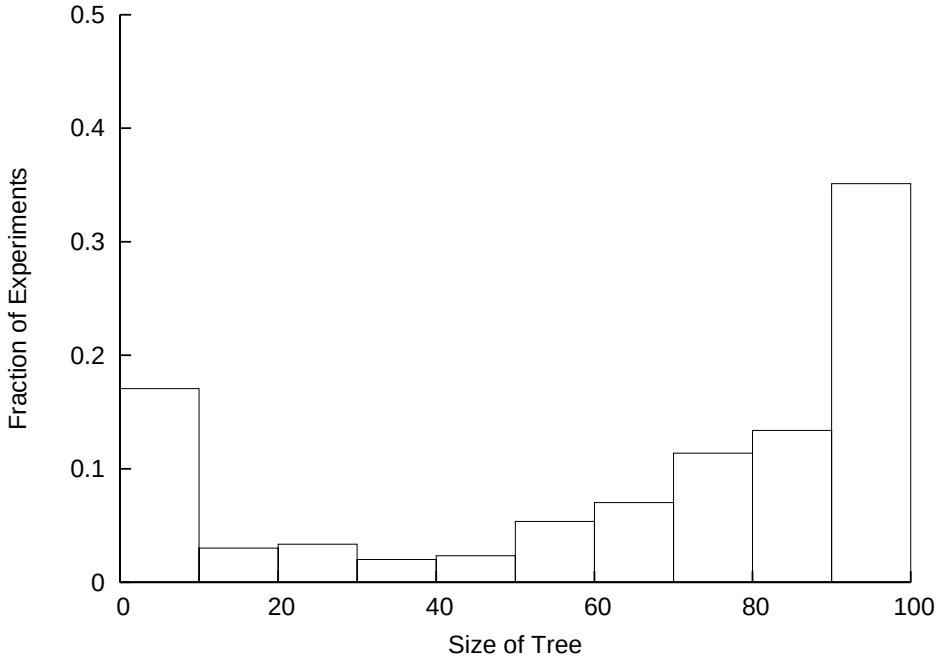
However, the plurality of scheduling problems are best scheduled by the reduced memory approximations encoded by the largest trees.

5.3.3 Comparative Evaluation of Decision Trees

We introduced the value-optimal scheduling problem in Chapter 3. Producing a value-optimal policy has large time complexity, and at run time this policy has large memory complexity. We investigated a variety of heuristics in Chapter 4, and found two, *greedy* (described in Section 4.2) and *Pseudo 0* (described in Section 4.4) to be best in the widest range of scenarios, as seen by the recommendations given in Section 4.6. We now compare reduced-memory tree-based approximations of the value-optimal policy to the other techniques in terms of their ability to maximize utility accrual. We also examine the effects of synthesizing our techniques, by examining the effect of selecting the best heuristic for a given problem instance and the effect of selecting (from among heuristics and tree-based approximations) the highest valued scheduling technique.



(a) Optimal tree size for soft real-time problem instances.



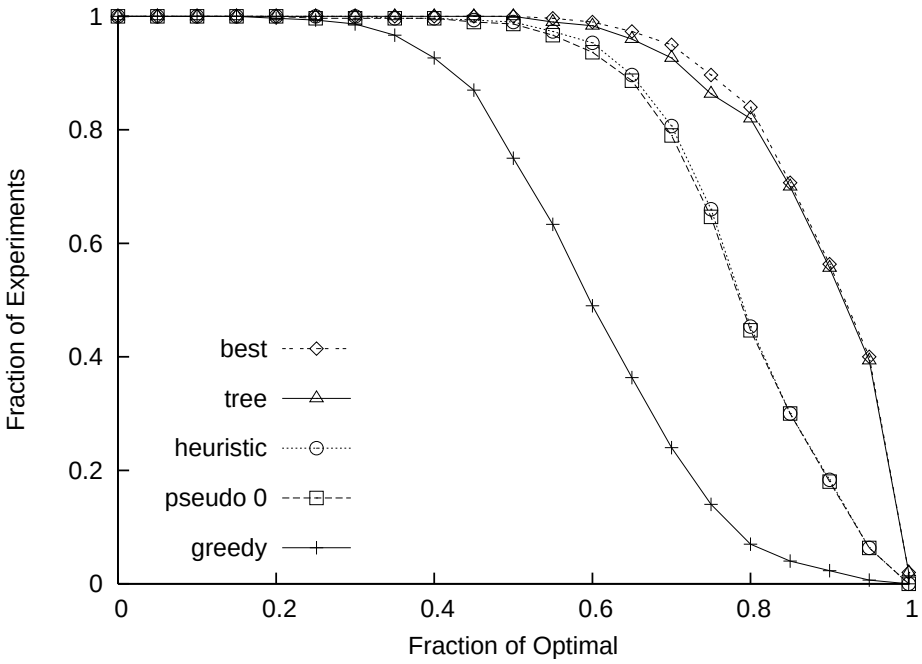
(b) Optimal tree size for hard real-time problem instances.

Figure 5.4: Histogram of the tree size giving the highest valued approximation.

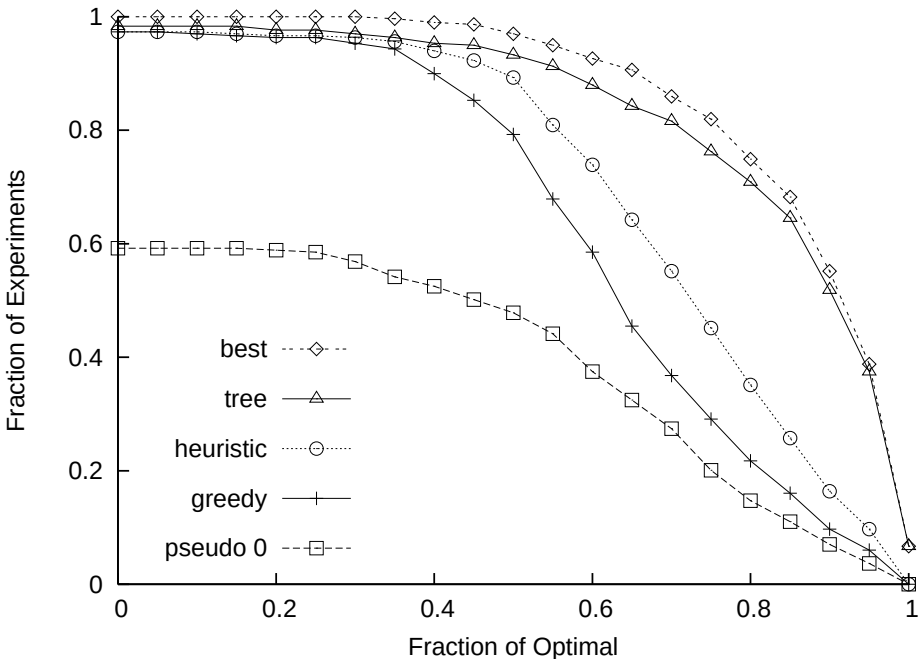
Figure 5.5 shows the value of the different scheduling techniques for the 300 scheduling problems introduced in Section 5.3.1. The greedy and Pseudo 0 heuristics are shown, as well as an evaluation (labeled *heuristic*) of the value achieved by simply picking which ever of these two heuristics performs best in each scheduling problem. As can be seen in Figure 5.5(a) this approach does not significantly improve the overall utility accrual in the soft real-time case. This is most likely because Pseudo 0 outperforms greedy in all but a few soft real-time scheduling problems. In contrast, as is shown in Figure 5.5(b), this approach does provide noticeable gains in the hard real-time case. This is likely because in the hard real-time scenarios, while greedy in general performs better, there may be a slightly greater chance that Pseudo 0 outperforms greedy on a particular scheduling problem.

The evaluation labeled *tree* gives the tree-based scheduling function with the highest value for all the tree sizes in the range [0,100]. In both scenarios, soft and hard real-time, our tree-based approximations outperformed either heuristic alone, as well as the approach of choosing the best heuristic for a given scheduling problem.

The evaluation labeled *best* is the technique of choosing the heuristic or tree-based approximation that gives the best value available for a given scheduling problem. As can be seen from Figure 5.5(a), there was little improvement over using the best tree-based approximation in the soft real-time scenario. However, in the hard real-time scenario shown in Figure 5.5(b) there was a more noticeable improvement. As before this improvement is likely due to the fact that in a hard real-time scenario there may be a slightly greater chance that a heuristic outperforms a tree-based approximation for that specific problem instance, when compared to the corresponding soft real-time scenario. Indeed, when we look at the percentage of time the *best* approach would choose a particular scenario, we find that under soft real-time scenarios, 83.3% percent of the time a tree-based approximation was used, while 17.7% of the time a heuristic was used (1.0% of instances were best scheduled by greedy, the other 16.7% by Pseudo 0). In the hard real-time scenarios, slightly fewer problems were best scheduled by a tree based approximation (78.7%), while 21.3% were best scheduled by a heuristic (13.7% of instances were best scheduled by greedy, the other 7.7% by Pseudo 0).



(a) Comparison of scheduling approaches for soft real-time problem instances.



(b) Comparison of scheduling approaches for hard real-time problem instances.

Figure 5.5: Evaluation of heuristics, decision trees and combined approaches.

5.4 Heuristic Leaf Nodes

In the trees evaluated in Section 5.3, we assumed that each leaf node was labeled with a single action. However, a final useful extension to our tree building algorithm is to allow leaf nodes to be labeled by a heuristic policy. This allows us to incorporate the strengths of heuristics in our tree-based approximations more directly and to synthesize utility-accrual policies with higher expected value, while still maintaining reasonable run-time complexity.

Given a set of policies Π and a set of state-action pairs $\mathcal{I}$ we calculate the sets,

$$\mathcal{I}_\pi = \{(x,a)|(x,a) \in \mathcal{I}, \pi(x) = a\} \tag{5.8}$$

and

$$\mathcal{I}_{-\pi,i} = \{(x,a)|(x,a) \in \mathcal{I}, \pi(x) \neq a, a = i\} \tag{5.9}$$

for each heuristic $\pi \in \Pi$. The first set is the set of states in the encoded policy that have the same state-action mapping as policy π. The other is the set of state-action pairs that do not have the same state-action mapping as π, but instead have the state mapped to action $i \in \mathcal{A}$. The entropy of a set relative to a policy $\pi \in \Pi$ is defined as:

$$h_\pi(\mathcal{I}) = -\frac{|\mathcal{I}_\pi|}{|\mathcal{I}|} \log_b \frac{|\mathcal{I}_\pi|}{|\mathcal{I}|} - \sum_{a \in \mathcal{A}} \frac{|\mathcal{I}_{-\pi,a}|}{|\mathcal{I}|} \log_b \frac{|\mathcal{I}_{-\pi,a}|}{|\mathcal{I}|} \tag{5.10}$$

Conceptually this measure can be seen as taking the original set $\mathcal{I}$, and replacing any action in a state-action pair where the encoded policy and policy π agree with a unique action for following the policy π, and then taking the entropy of the resulting set. It is worth noting that this definition of entropy is identical to the measure of entropy given in Equation 5.4 if a special policy is considered that returns a single static action for any state.

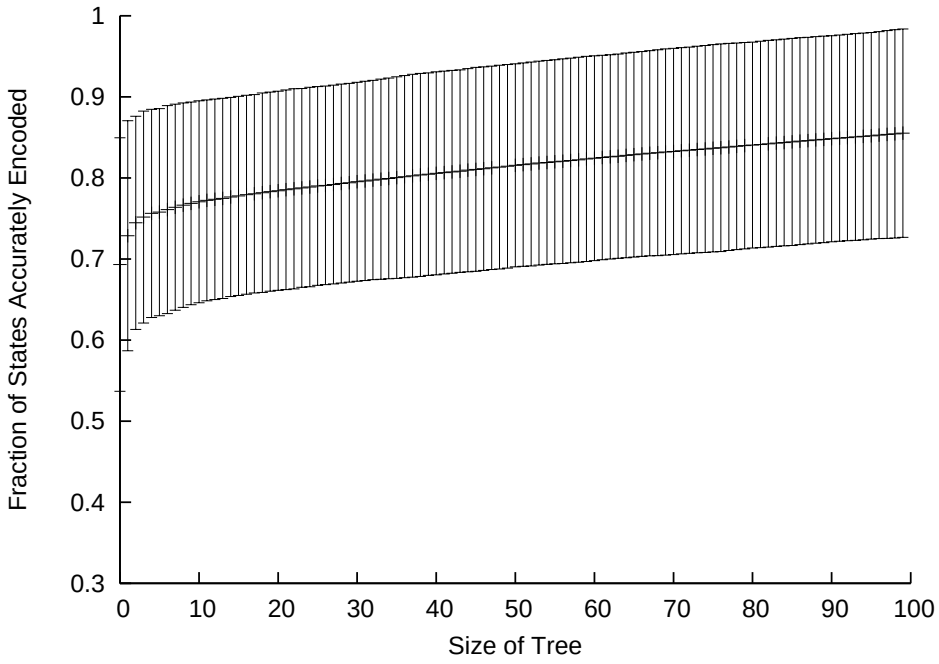
When considering both static actions and a set of policies Π, the entropy of a set is:

$$\min\{h(\mathcal{I}), \min_{\pi \in \Pi}\{h_\pi\}\} \tag{5.11}$$

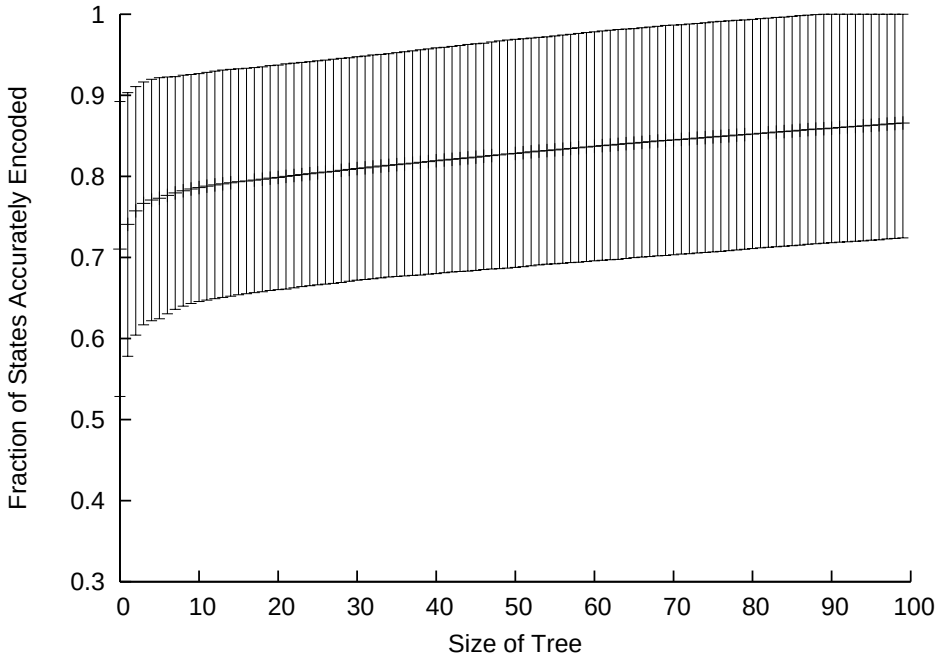
This formulation takes into account that the label of any leaf node will be either the policy or static action with the largest set I_π or I_i.

We show the results of running the experiments in Section 5.3 for trees with heuristic leaves in Figure 5.6. These results demonstrate that in this approach the encoding accuracy for trees with heuristic leaves begins slightly higher for very small trees, but quickly becomes indistinguishable from trees built solely with static actions. Figure 5.7 shows that in this approach the optimally-valued trees with heuristic leaves was more likely to be small (under 10 splits), but otherwise the results were not significantly different than the results for the static trees shown in Figure 5.4.

Figure 5.8 shows that decision trees with heuristic leaves also outperform the heuristic policies overall in both hard and soft real-time problem instances. For soft real-time experiments, 95.3% of the time the best scheduler was a tree with heuristic leaves, while an additional 4.3% of the time the best tree with heuristic leaves was equivalent to the best heuristic (typically a "tree" with only a single node labeled with the heuristic). In only 0.3% of cases was the best solution a heuristic. For hard real-time experiments, 88.3% of the time the best scheduler was a tree with heuristic leaves, while an additional 8.4% of the time the best tree with heuristic leaves was equivalent to the best heuristic. In 3.3% of cases the best solution was a heuristic (2.3% of the time greedy, 1.0% of the time Pseudo 0).

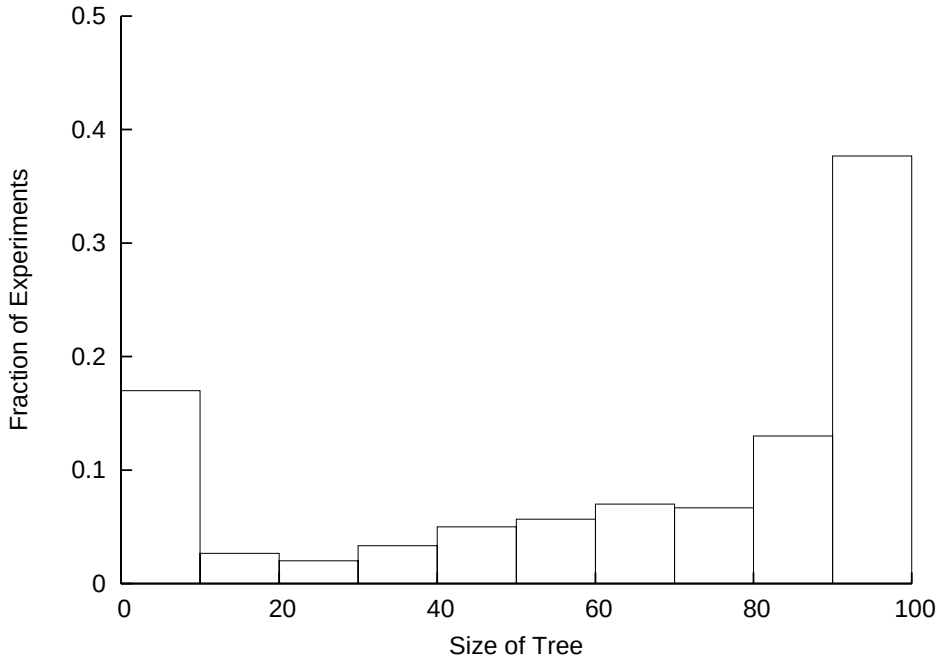


(a) Compression of 5-task soft real-time policies.

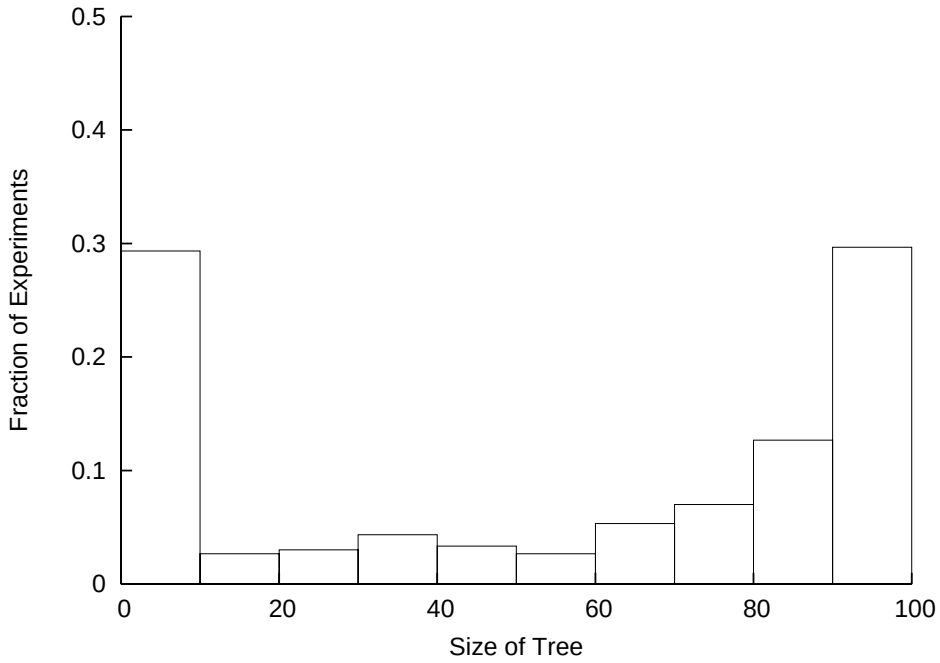


(b) Compression of 5-task hard real-time policies.

Figure 5.6: Accuracy of policy encoding as a function of the size of the tree with heuristic leaves, counted as the number of splits. Average accuracy with 95% confidence intervals, based on all 300 problem instances, is shown.

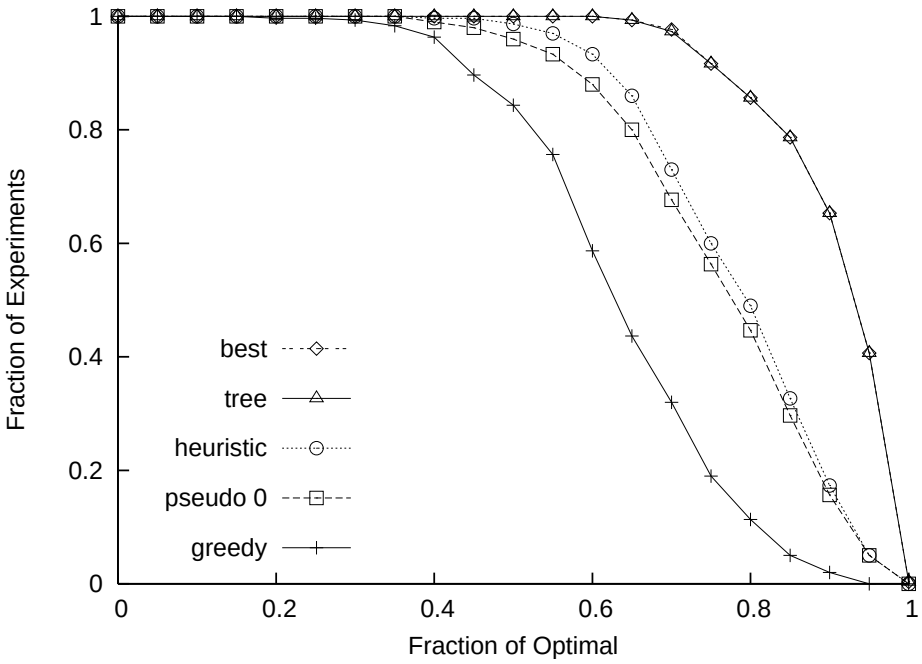


(a) Optimal tree size for soft real-time problem instances.

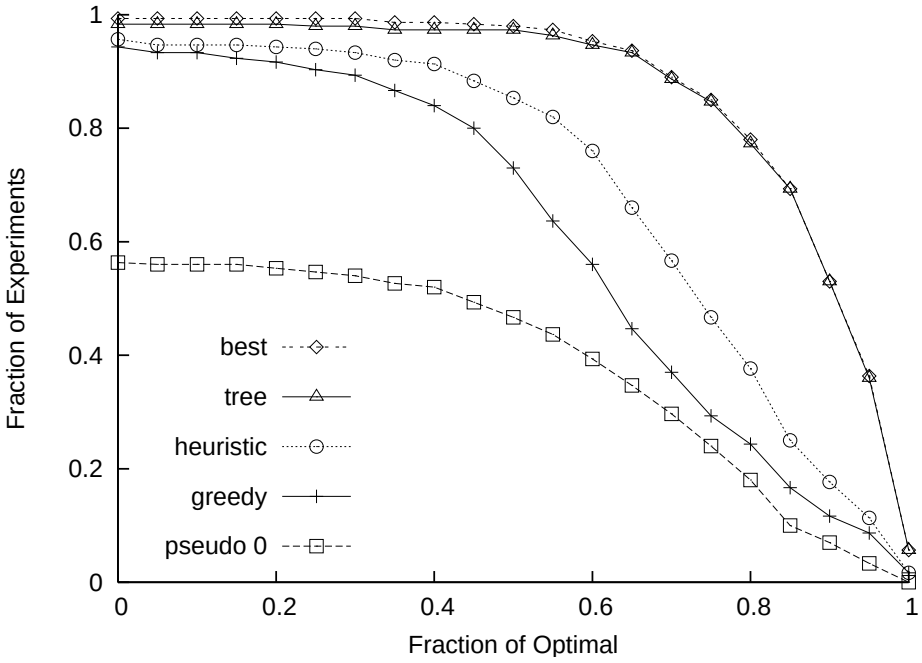


(b) Optimal tree size for hard real-time problem instances.

Figure 5.7: Histogram of the size of the tree with heuristic leaves that gives the highest valued approximation.



(a) Comparison of scheduling approaches for soft real-time problem instances.



(b) Comparison of scheduling approaches for hard real-time problem instances.

Figure 5.8: Evaluation of heuristics, decision trees, and combined approaches.

Chapter 6

Conclusions

Time utility functions are a promising scheduling abstraction for an emerging class of real-time and cyber-physical systems, whose complex timing constraints may be difficult to represent by traditional scheduling abstractions such as deadlines. However, the design of utility-aware scheduling algorithms poses a number of important research challenges.

This dissertation introduces a novel Markov Decision Process (MDP) model for systems with periodic tasks and non-preemptive jobs that run with stochastic duration. This model allows us to quantify the exact value of utility-accrual scheduling policies for these systems, even in the face of stochastic behavior and rare high impact events.

We can derive value-optimal scheduling policies, policies that maximize long term expected utility accrual, by solving the MDP system model. However, this incurs a high up-front cost in time and memory, as well as a significant run-time memory cost.

Because of the potentially prohibitive cost associated with value-optimal schedules we consider a variety of utility-aware scheduling heuristics that offer low time and memory complexity at run-time. Using our MDP system model, we can examine how each heuristic compares to the value-optimal policy. We examined the effect of soft versus hard real-time constraints, as well as the effect of load, and the effect of different time utility function types. We introduced a set of recommendations for which heuristics are most effective in different scenarios, though in some cases none of the heuristics performed adequately.

We then introduced tree-based approximations of value-optimal policies, which trade off some value-optimality for significantly reduced memory complexity. We compared the value of the resulting reduced memory approximation policies to value-optimal and heuristic policies. Our results show that the tree-based approximations often outperform heuristics. Finally, we show the effect of synthesizing heuristics and tree-based approximations. Our results show that through a combination of the techniques presented in this dissertation, it is possible to synthesize utility accrual schedulers with low run-time complexity that compare favorably to value-optimal policies.

There are several open research questions that were uncovered as part of this dissertation. First, there is the unresolved question of why the sequencing heuristic (discussed in Chapter 4) performs comparatively poorly, especially when a simplified variation of sequencing improves performance when applied by UPA α. Second, it is unclear if there is an efficient or even effective algorithm for calculating the optimal value for α for UPA α and Pseudo α in hard real-time scenarios. Third, the ability of decision tree based schedulers to adapt to unforeseen system states remains a largely unexplored feature of this approach. The effect of unexpected states on the value of scheduling policies is unknown. These questions remain open for future work.

There are several other promising research directions that future work could explore. Some of these potential extension are the application of techniques from this work to other system and task models e.g. tasks with dependencies or sporadic tasks. Other avenues for future work include examining the ability to adapt to a changing system, especially in domains such as supply-chain management in which the time-scale of the system makes offline recomputation of the value-optimal in response to changing system conditions tractable.

References

[1] T. F. Abdelzaher. An Automated Profiling Subsystem for QoS-Aware Services. In *Proceedings of the Sixth IEEE Real-Time Technology and Applications Symposium*, pages 208–217, Washington DC, USA, 2000.

[2] A. Atlas and A. Bestavros. Statistical Rate Monotonic Scheduling. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 123–132, Madrid, Spain, 1998.

[3] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[4] C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic Dynamic Programming with Factored Representations. *Artificial Intelligence*, 121(1–2):49–107, 2000.

[5] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2005.

[6] G. Buttazzo and E. Bini. Optimal Dimensioning of a Constant Bandwidth Server. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 169–177, Rio de Janeiro, Brazil, 2006.

[7] K. Chen and P. Muhlethaler. A Scheduling Algorithm for Tasks Described by Time Value Function. *Real-Time Systems*, 10(3):293–312, 1996.

[8] R. Clark, E. D. Jensen, A. Kanevsky, J. Maurer, P. Wallace, T. Wheeler, Y. Zhang, D. Wells, T. Lawrence, and P. Hurley. An Adaptive, Distributed Airborne Tracking System. In *IEEE Workshop on Parrallel and Distributed Real-Time systems*, volume 1586 of *LNCS*, pages 353–362. Springer-Verlag, 1999.

[9] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, 1990.

[10] Y. Fu, N. Kottenstette, Y. Chen, C. Lu, X. Koutsoukos, and H. Wang. Feedback Thermal Control for Real-time Systems. In *Proceeding of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 111–120, Stockholm, Sweden, 2010.

[11] C. D. Gill, D. C. Schmidt, and R. K. Cytron. Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing. *IEEE Proceedings, Special Issue of Modeling and Design of Embedded Systems*, 91(1):183–197, 2003.

[12] R. Glaubius. *Scheduling Policy Design using Stochastic Dynamic Programming*. PhD thesis, Washington University in St. Louis, 2010.

[13] R. Glaubius, T. Tidwell, C. Gill, and W. D. Smart. Scheduling Design with Unknown Execution Time Distributions or Modes. Technical Report WUCSE-2009-15, Washington University in St. Louis, 2009.

[14] R. Glaubius, T. Tidwell, C. Gill, and W. D. Smart. Scheduling Policy Design for Autonomic Systems. *International Journal on Autonomous and Adaptive Communications Systems*, 2(3):276–296, 2009.

[15] R. Glaubius, T. Tidwell, B. Sidoti, D. Pilla, J. Meden, C. Gill, and W. D. Smart. Scalable Scheduling Policy Design for Open Soft Real-Time Systems. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Application Symposium*, pages 237–246, Stockholm, Sweden, 2010.

[16] R. Glaubius, T. Tidwell, W. D. Smart, and C. Gill. Scheduling Design and Verification for Open Soft Real-Time Systems. In *Proceedings of the 29th Real-Time Systems Symposium*, pages 505–514, Barcelona, Spain, 2008.

[17] R. Guerra and G. Fohler. A Gravitational Task Model with Arbitrary Anchor Points for Target Sensitive Real-Time Applications. *Real-Time Ssytems*, 43(1):93–115, 2009.

[18] C. Guestrin, D. Koller, R. Parr, and S. Venkataraman. Efficient Solution Algorithms for Factored MDPs. *Journal of Artificial Intelligence Research*, 19:399–468, 2003.

[19] D. Isovic, G. Fohler, and L. Steffens. Timing Constraints of MPEG-2 Decoding for High Quality Video: Misconceptions and Realistic Assumptions. In *Proceeding of the 15th Euromicro Conference on Real-Time Systems*, pages 73–82, Porto, Portugal, 2003.

[20] E. D. Jensen, C. D. Locke, and H. Tokuda. A Time-Driven Scheduling Model for Real-Time Systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium*, pages 112–122, San Diego, CA, USA, 1985.

[21] P. H. Jones. *Optimizing Application Performance Using Temperature Feedback*. PhD thesis, Washington University in St. Louis, 2008.

[22] L. Kaelbling, M. Littman, and A. Moore. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[23] N. Kohl and P. Stone. Machine Learning for Fast Quadrupedal Locomotion. In *Proceedings of the 19th National Conference on Artificial Intelligence*, pages 611–616, San Jose, CA, USA, 2004.

[24] P. Li. *Utility Accrual Real-Time Scheduling: Models and Algorithms*. PhD thesis, Virginia Tech, 2004.

[25] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River, New Jersey, USA, 2000.

[26] C. D. Locke. *Best-Effort Decision-Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986.

[27] C. L. Lui and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[28] N. Manica, L. Abeni, and L. Paloponi. Reservation-based Interrupt Scheduling. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Application Symposium*, pages 45–55, Stockholm, Sweden, 2010.

[29] S. Manolache, P. Eles, and Z. Peng. Schedulability Analysis of Applications with Stochastic Task Execution Times. *ACM Transactions on Embedded Computing Systems*, 3(4):706–735, 2004.

[30] P. Marti, G. Fohler, K. Ramamritham, and J. M. Fuertes. Jitter Compensation in Real-Time Control Systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 39–48, London, UK, 2001.

[31] A. A. Martinez and P. Tabuada. On the Benefits of Relaxing the Periodicity Assumption for Networked Control Systems over CAN. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 3–12, Washington DC, USA, 2009.

[32] A. F. Mills and J. H. Anderson. A Stochastic Framework for Multiprocessor Soft Real-Time Scheduling. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Application Symposium*, pages 311–320, Stockholm, Sweden, 2010.

[33] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[34] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang. Inverted Autonomous Helicopter Flight via Reinforcement Learning. In *Proceedings of the 9th International Symposium on Experimental Robotics*, volume 21 of *Springer Tracts in Advanced Robotics*, pages 363–372. Springer, 2004.

[35] A. Y. Ng, H. J. Kim, M. I. Jordan, and S. Sastry. Autonomous Helicopter Flight via Reinforcement Learning. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, pages 799–806. MIT Press, Cambridge, MA, USA, 2004.

[36] R. Pellizzoni, B. Bui, M. Caccamo, and L. Sha. Coscheduling of CPU and I/O Transactions in COTS-Based Embedded Systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 221–231, Barcelona, Spain, 2008.

[37] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, 2005.

[38] M. L. Puterman and M. C. Shin. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24:1127–1137, 1978.

[39] J. R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1(1):81–106, 1986.

[40] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[41] B. Ravindran, E. D. Jensen, and P. Li. On Recent Advances in Time/Utility Function Real-Time Scheduling and Resource Management. In *Proceeding of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 55–60, Seattle, WA, USA, 2005.

[42] P. K. Saraswat, P. Pop, and J. Madsen. Task Mapping and Bandwidth Reservation for Mixed Hard/Soft Fault-Tolerant Embedded Systems. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Application Symposium*, pages 89–98, Stockholm, Sweden, 2010.

[43] W. D. Smart and L. P. Kaelbling. Practical Reinforcement Learning in Continuous Spaces. In *Proceedings of the 17th International Conference on Machine Learning*, pages 903–910, San Francisco, CA, USA, 2000.

[44] J. A. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, S. Son, and C. Lu. Feedback Control Scheduling in Distributed Real-Time Systems. In *Proceedings of the 23nd IEEE Real-Time Systems Symposium*, pages 59–69, London, UK, 2001.

[45] J. A. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of Classical Scheduling Results For Real-Time Systems. *IEEE COMPUTER*, 28:16–25, 1995.

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

[47] T. Tidwell, C. Bass, E. Lasker, M. Wylde, C. D. Gill, and W. D. Smart. Scalable Utility Aware Scheduling Heuristics for Real-Time Tasks with Stochastic Non-Preemptive Execution Intervals. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pages 238–247, Porto, Portugal, 2011.

[48] T. Tidwell, R. Glaubius, C. Gill, and W. D. Smart. Scheduling for Reliable Execution in Autonomic Systems. In *Proceedings of the 5th International Conference on Autonomic and Trusted Computing*, volume 5060 of *LNCS*, pages 149–161. Springer-Verlag, 2008.

[49] T. Tidwell, R. Glaubius, W. D. Smart, and C. Gill. Optimizing Expected Time Utility in Cyber-Physical System Schedulers. In *Proceedings of the 31st Real-Time Systems Symposium*, pages 193–201, San Diego, CA, USA, 2010.

[50] J. Wang and B. Ravindran. Time-Utility Function-Driven Switched Ethernet: Packet Scheduling Algorithm, Implementation, and Feasibility Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 15(1):119–133, 2004.

[51] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.