A Scheduling Analysis Tool for Real-Time Systems

by

Clarence Bruce Applegate

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000

Author ....
Department of Electrical Engineering and Computer Science
May 24, 2000

Certified by ....
Dr.Amar Gupta
Co-Director, PROFIT Program
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Scheduling Analysis Tool for Real-Time Systems

by

Clarence Bruce Applegate

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2000, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Masters of Engineering in Computer Science and Engineering

Abstract

Real-time analysis is needed for optimizing the use of many real-time systems.There are a wide variety of scheduling policies, based on different sets of criteria, designed to optimize the use of these systems.The focus of this project is on optimizing the utility of a schedule, where the utility of a schedule is a function of a set of utility functions describing the time-dependent utility of completing each computation.For this project, a module for utility-based real-time system scheduling was created.This module contains a scheduler that implements a greedy utility-based scheduling policy and an analysis function that compares the utility of the schedule generated to an approximation of the maximum achievable utility.

Thesis Supervisor: Dr.Amar Gupta
Title: Co-Director, PROFIT Program

Acknowledgments

First and foremost, to Dr.Amar Gupta. Without his support and guidance I would have strayed from this goal a long time ago. His interest, advice, and encouragement have been invaluable in all of my academic decisions as long as i've worked with him.

To the staff at MITRE for giving me this opportunity and letting me make what I could of it. I would especially like to thank Doug Jensen for his insight and guidance; Tom Wheeler, for making sure that I had everything I needed; and Ray Clark, my direct supervisor, who gave me clear direction when I was lost, was always available when needed, and gave me the freedom to create while always making sure I was on the right track.

To both TimeSys and TriPac for their support in my use of their products in this work. Especially to Srini Vasan, Doug Locke, and Raj Rajkumar at TimeSys for clearing up every question that I had as quickly as possible. They kept this whole project running smoothly.

To Alex Raine and William Chuang, who answered my stupid C++ questions at any strange hour.

To my girlfriend, Victoria Keith, who stuck with me, watched over me, and helped me through this even though it ate up what otherwise would have been our time together.

To my parents, who have given me such incredible love and support: you make me want to make you proud. I hope I have.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real-time analysis is applicable in numerous real-world situations. It is useful when a system has a limited number of resources, limited time in which to use these resources, or any other constraints in which the functioning of the system can be improved by scheduling. The focus of this project is processor scheduling, although the methodology could apply to any resource with the same sort of information and requirements. Processor scheduling deals with the allocation of available processor cycles. The processor[1] is a finite resource and can be utilized by at most one task at any point in time. In addition, each task must satisfy a certain set of constraints, such as execution time and precedence constraints. Thus, the processor scheduling problem can be described as assigning a number of tasks to execute on a processor in such a way that the task constraints are met and some criteria are optimized.

1.1 Optimization Criteria

There are numerous types of constraints and criteria for optimization that vary depending on the system being studied. These constraints may be based on such factors as time, utility, and available resources.

[1]The general processor scheduling problem can include multiple processors. For the purposes of this document, however, only the single processor scheduling problem is being considered.

1.1.1 Temporal Criteria

Real-time systems include tasks that have time constraints. One common example is a *deadline*, where the task must finish by a certain time to have maximal value to the system. There are two main types of deadlines: *hard deadlines* and *soft deadlines* [13]. Hard deadlines are deadlines that require the task to complete by the deadline to be of any value to the system. Soft deadlines, on the other hand, have a certain value if they complete by the deadline but have a lesser, non-zero value if they complete after the deadline.

Deadlines are actually a special case of a more general class of functions, also known as utility functions. A utility function describes the usefulness of completing the task with respect to time of completion. Under one formulation, the goal of real-time systems with respect to time-based criteria is for their tasks to be scheduled such that they complete at acceptable times with acceptable *predictability* [13], [15].

A *schedule of tasks* is a temporal sequence in which the tasks utilize the available resources. Thus, acceptable collective timeliness can be described as a schedule satisfying certain criteria. For example, in the case of scheduling hard real-time systems, the criterion is to meet all hard deadlines. For soft real-time, there are a variety of possible criteria, including minimizing number of missed deadlines, minimizing average *lateness* or *tardiness*, and maximizing the total utility. Lateness is defined as the difference between the time of completion of a task and its deadline. This value can be negative if the time of completion is before the deadline, so the concept of tardiness is often more useful. Tardiness is the maximum of lateness and zero or, in other words, the lateness only if a job finishes execution after its deadline [11].

Predictability describes the degree to which something is known in advance, ranging from pure determinism to complete entropy. For hard real-time, a high enough degree of predictability is necessary to guarantee a task will complete on time. The degree of predictability necessary for hard deadlines depends very much on the parameters of the specific problem. For example, scheduling a resource where the resource needs to be in use constantly to meet the criteria would need to have a high degree of predictability, but if the resource only had sporadic usage the predictability might not need to be as high. Soft real-time can encompass varying degrees of predictability, sometimes implicitly stated. For example, a periodic task implicitly occurs with high predictability once per period. Lower predictability for soft real-time may lead to some deadlines not being met. However, not meeting a deadline in soft real-time does not carry quite the import of not meetin a deadline in hard real-time.

Tasks have two additional temporal attributes, the *release time* and the *execution time*, which are highly relevant in any implementation of a real-time scheduling policy. The release time, or ready time, is the time that the task first lets the machine know of its presence and becomes available for execution on the machine [11]. The execution time is the amount of time the task must actually utilize the processor before it completes. In some cases, the execution time is a *stochastic* variable. A stochastic variable is one that cannot be described by a deterministic fixed number; it is often described by a probabilistic distribution. In the case of stochastic variables, this work considers the execution time to be the worst case execution time [11]. This choice may not be desirable in all cases (like in best-case scenarios). It was chosen as an expedient for this work.

1.1.2 Priority

Real-time systems also often have tasks that need to be scheduled based on some measure of value or importance. One common approach to this is to assign a *priority* or *weight* to each task. A priority is a ranking of importance; a task with higher priority takes precedence over those with lower priorities. This is a very common concept in processor scheduling. For example, when an interrupt is sent to a processor the processor often needs to handle it more urgently than whatever it may be currently running. A scheduling policy that considers the interrupts as a higher priority than other tasks can handle this kind of situation. Priority-driven scheduling has been implemented in such applications as processor and microkernel scheduling [18].

1.1.3 Machine Complexity

Real-time systems deal with different levels of complexity within the machines themselves. For one, there may be multiple machines on which to schedule the jobs. This is the case considered in the general job-shop problem. Systems that operate in parallel need to be able to consider multiple machines running simultaneously when generating the schedule [20]. In addition, a single machine may be able to utilize multiple *resources*. A resource is some component of the system that can be utilized by a machine while a task is executing on the machine. Resources can be shared by all machines or only available to a single machine. For example, consider a printer on a computer network. The printer can only print out one job at a time, so that particular resource must be scheduled for all machines in the system. For this set of problems, additional constraints may be derived from the quality of usage of these resources. Some of the possible constraints include load balancing and

1.2 Terminology

There are additional scheduling considerations that must be introduced to fully describe relevant scheduling problems. They are different aspects of scheduling problems that occur in various systems. The first idea is that of *preemption*. When a task is executing on a machine, another task may reach its release time and become available to the machine. In some cases, the machine will achieve an optimal result only if it stops execution on the currently executing task and allows the newly available task to begin executing. This is known as preemption [4].

The question then becomes what the preempted task is allowed to do, relative to its own completion. The two extremes are *preempt-resume* mode and *preempt-repeat* mode. In preempt-resume mode, the task is allowed to begin executing where it left off, so there is no additional execution time taken by the task itself if it completes execution. On the other hand, in preempt repeat mode if a task is preempted it must begin again as if it had not executed at all [4].

In the case of an infeasible schedule, it is necessary to *shed* various tasks. Shedding halts execution of a task and removes it from consideration for future execution. One example is the removal of the task from an *execution queue.* An execution queue can be defined as a queue containing all currently executable tasks and the order in which they will be executed, although the order may change. For many implementations, it is necessary to consider more complicated situations, such as what occurs when a partially completed task holds a shared resource. For this resource, however, there is only one resource and one processor, so the definition above is sufficient for this scope.

There are many systems in which it is necessary to consider dependencies. Dependencies are relations between tasks where what can and does occur with one task or set of tasks determines what can occur with another task or set of tasks. The definition is narrowed for the scope of this project. For this case, dependencies are relations between different tasks where, if task_i is dependent on task_j, task_j must complete execution before task_i begins running. Dependencies can have multiple levels, so task_i can depend on task_j, which depends on task_k.

Chapter 2

Scheduling Policies and Analysis

There are a wide variety of scheduling policies and forms of analysis that have been developed, based on different environments (e.g., single or multiple machines, single or multiple resources available for the machines), timing considerations (e.g., presence of dependencies, preemption, task shedding, periodicity of events), assumptions about information (e.g., integrality of data, release times at zero), and optimization criteria (e.g., timing, priority) [20], [9], [4]. This project deals with a single processor/single resource. It only considers single occurrence events (where a single occurrence event only happens once per simulation), although it is possible to deal with periodic events by representing them as a series of single occurrence events that are periodically spaced. There are a variety of algorithms that are relevant for the purposes of this project.

2.1 Scheduling Policies

2.1.1 Earliest Deadline First

Earliest Deadline First (EDF), or Earliest Due Date, is an algorithm developed by J.R.Jackson that will produce an optimal schedule based on certain timeliness criteria [12]. The deadline is the time by which the task must be completed. The algorithm schedules tasks in deadline order, from the task with the earliest deadline to the task with the latest deadline. This algorithm guarantees that it will either produce a feasible solution or guarantee that no feasible solution exists. The running time of this algorithm is O(n log n) time, where n is the number of jobs that need to execute [10].

The original EDF algorithm assumes that release times and deadlines are integral, and all processing times are one unit long. In addition, it assumes no preemption is allowed. The problem was initially described for a single processor, but it can be extended to multiple machines by a simple round-robin assignment: given m machines, the first m tasks are assigned to machines 1,...m, the next m tasks are then assigned to machines 1,...,m, etc. [10]. The implementation is as described below.

EDF

```
For t $\leftarrow$ 0 to T (T = total processor time available in schedule)
    Look at all jobs $j_1,...,j_i$ with release times $\leq$ t and deadlines $t_1,...,t_i$
    $j' \leftarrow j_1$
    $t' \leftarrow t_1$
    For k $\leftarrow$ 1 to i
        if $t_k < t'$
            $j' \leftarrow j_k$
            $t' \leftarrow t_k$
        if $t_k > t$
            FAIL
```

The problem was initially described for a single processor, but it can be extended to multiple machines by a simple round-robin assignment: given m machines, the first m tasks are assigned to machines 1,...m, the next m tasks are then assigned to machines 1,...,m, and similarly for all remaining tasks [10]. The difficulty arises when considering integral, rather than unit, processing times. In this case, the problem is actually an NP-complete problem [10]. However, in these cases, EDF might act as a reasonable heuristic. The EDF approach has been shown to minimize both the maximum task lateness and the maximum task tardiness. These results remain consistent if dependencies are considered as well [4].

A variant of EDF, called EDF-var, of this is employed as one of the scheduling policies available in the scheduling and analysis module implemented for this project. EDF-var has a few features that differ from the problem as described above. For one, EDF-var allows for preemption. If preemption is introduced into EDF scheduling, the NP-complete problem becomes much more tractable. In addition, the release times, execution times, and even time in simulation are not guaranteed to be integral. Finally, EDF-var sheds tasks that do not complete in time. This is because the optimization criteria of the scheduler do not include minimizing lateness or tardiness, and a task that does not complete in time has no value to the system.

2.1.2 Shortest Processing Time

Shortest Processing Time (SPT) is another common scheduling algorithm. In this algorithm, the objective is to minimize the mean *flow time*. The flow time is defined as the difference between the release time and the completion time. If it is assumed that there is a single machine, integrality of data (all temporal data can be described as integer values), release times at start of simulation, and neither dependencies nor preemption, executing tasks in non-decreasing order of execution time minimizes the mean flow time [5], [9].

2.1.3 Priority-Based Scheduling

Priority scheduling rules are actually a more global phenomenon that can also be used to describe time-based scheduling rules. For this case, priority-based scheduling is considered scheduling based on some measure of priority or value that is independent of other relevant scheduling characteristics (e.g., execution time, deadline). It is possible to use execution time or deadline-based priority rules, but simply for the case of keeping the policies distinct they are considered separately in this case. For priority-based schedulers, the general policy is to first schedule tasks with the highest priority. Once that has been accomplished, schedule the tasks with the next highest priority, and so on. Priority-based schedulers will often use another scheduling policy when deciding to schedule among those with equivalent priority (e.g., FIFO, Round Robin). [17]

2.2 Analysis

2.2.1 Rate Monotonic Analysis

The most common form of analysis available today is Rate Monotonic Analysis (RMA), invented in the 1970's by Liu and Leyland [19]. RMA is arguably aimed toward scheduling periodic functions. It is a pessimistic analysis tool that considers both periodicity and worst-case execution times to determine if a set of tasks is schedulable. It has been successful in practice, as it is the analysis approach that is basis of multiple commercial scheduling and analysis products, such as TimeWiz [25] and RapidRMA [26].

RMA guarantees that a schedule is feasible if the total task utilization is less than or equal to a utilization bound. For a set of N tasks, the total task utilization is defined as $\sum_{i=1}^{N} \frac{C_i}{T_I}$, where C_i is worst-case execution time of task$_I$, and T_i is the length of task$_i$'s period. The fraction $\frac{C_i}{T_I}$ is effectively the percent of each period that task$_I$ uses. The utilization bound U(N) is defined as $U(N) = N(2^{\frac{1}{N}} - 1)$. RMA states that if the utilization bound is greater than or equal to the total task utilization, then the set of tasks can always be scheduled [26], [19], [16].

This algorithm is inappropriate for the needs of this project. Most importantly, RMA determines whether or not a set of processes is schedulable. This is important in many contexts, but the aim of utility-based scheduling is to find the best schedule whether or not all tasks are schedulable. In fact, there are cases in which all the processes are schedulable but the highest utility schedule sheds some of the processes. For example, consider the following case:

- Two tasks t_1 and t_2 running on a single processor
- Each task requires 5 units of execution time

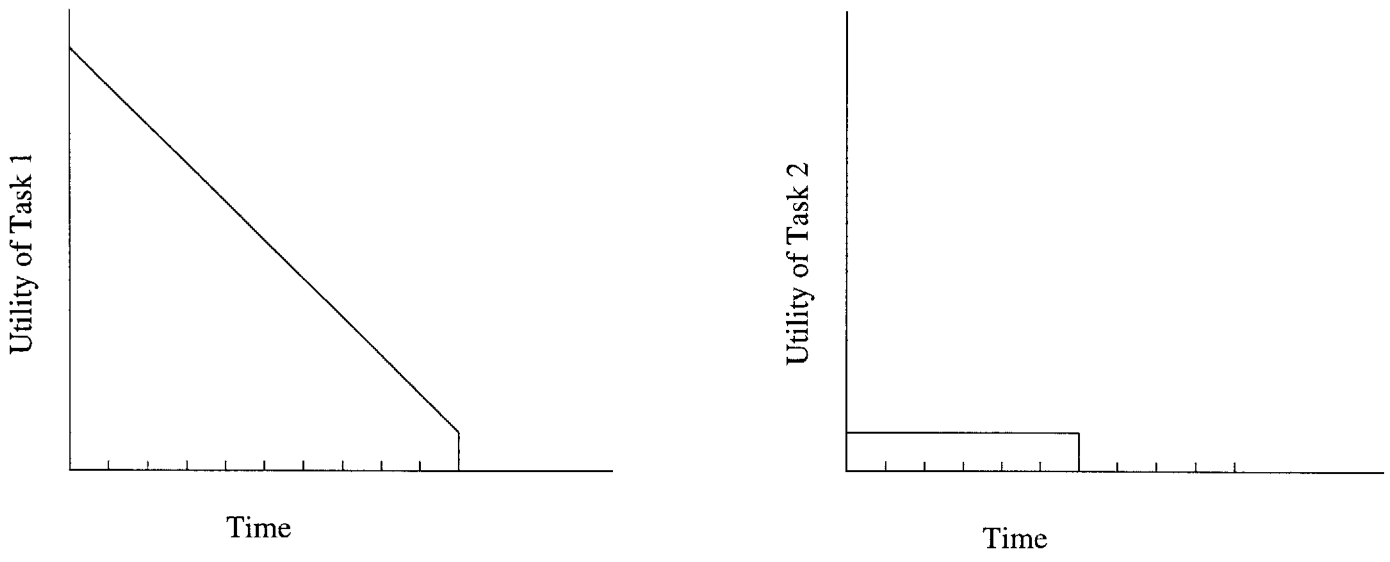

Figure 2-1: Utility functions of t_1 and t_2

- Task t_1 has a utility function described by a line that starts at time 0 with utility 11 and ends at time 10 with utility 1
- Task t_2 has a utility function described as a constant value of 1 from time 0 to time 6 and 0 until time 10 (i.e., a hard deadline at time 6)

If t_2 executes from 0 to 5 and t_1 executes from 5 to 10, both tasks complete. However, the total utility of this schedule is just the sum of utilities at their completion times, which ends up being a utility of 2. On the other hand, if t_1 executes from 0 to 5 and t_2 is shed, the utility of the schedule ends up being 6 even though t_2 was never executed.

In addition, RMA generally assumes a set of periodic tasks; for the case of the utility-based scheduler the desired set of tasks might not be predominantly periodic. There has been some work to extend RMA to tasks that are not purely periodic [16]. Even with this work, RMA is not particularly well suited towards the analysis of the systems we are considering. Finally, not only does RMA not consider utility functions, it does not even consider any kind of priority. For these reasons a new form of analysis must be developed.

Chapter 3

Problem Description

The goal of this project is to provide an analysis tool that performs utility-based analysis on a model of a real-time system. The analysis tools that currently exist are designed to analyze systems with the rate monotonic analysis approach. Although these tools are not capable of handling utility-based scheduling in their original form, it is possible to adapt them for the purposes of utility-based scheduling. In fact, one of these tools (TimeWiz [25]) was adapted for this implementation by adding a utility module, informally called UtilityWiz.

The utility-based scheduling problem is not as simple as it first may seem. The systems in question are sufufficiently complex and dynamic that formulae and approaches as simple as rate monotonic analysis are insufficient to address the problem. New combinations of rules and simulations must be developed for this analysis.

The expresiveness of utility functions allows them to cover a wide variety of scheduling issues. This has the advantage of covering most of the previous work, plus much more. The disadvantage, however, is that the level of complexity and depth of the problem effectively require that a heuristic approach be taken.

UtilityWiz supports dynamic event arrival and utility functions. This tool needed to cover these issues to achieve meaningful results. In addition, other issues such as allowing for dependencies and the ability to analyze not only the results but also the quality of the analysis, both in accuracy and running time, greatly augment the capabilities of this tool.

These features are not currently available on any extant commercial evaluation tool. Two companies that have built analysis tools (TriPacific Software Inc. [26] and TimeSys Corporation[25]) had previously expressed interest in adding this functionality to the current capabilities of their products but have lacked the resources to this point to undertake the task. The tool created in this project is implemented as a plug-in to one of their products.[1]

3.1 Utility Accrual Models

There are a variety of possible models for determining the utility of a schedule. For example, the utility of a schedule could be based partially on whether all tasks complete. In this case, a schedule where all tasks complete could be worth more than the case where not all complete but those that complete have a much higher utility. Another model would only accrue utility from a set of dependent tasks if all the tasks in the set complete. The model considered for this implementation is a simple utility accrual model that has been used in previous work[23],[7]. In this model, the utility of a schedule is simply the sum of the utilities of all the tasks at their times of completion.

3.2 Problem Statement

The utility-based scheduling optimization problem can be described as follows:

Given:

- a set of tasks 1,..,n, where each task i has:
  - a time-dependent utility function $u_i(t)$
  - an execution time e_i
  - a release time s_i

[1]This work could have been based on either company's product. The selection of TimeWiz does not indicate any believed superiority to RapidRMA.

  - an optional dependency d$_i$:

$$d_i = \begin{cases} \text{j, i must complete before i starts} \\ \emptyset\text{, there is no such j} \end{cases}$$

- a single processor on which to execute the tasks
- a time interval [0,T]

Define a schedule S as a set of times $t_i \forall$ tasks i, such that:

- t_i is the completion time of i if finishes execution in [0,T]
- t_i is $\emptyset$ if i does not finish execution in [0,T]

Define a legal schedule as a schedule such that:

- at any time t' $\in$ [0,T], no more that one task is utilizing the processor
- every task i:
  - executes for a total time of $\leq e_i$
  - does not begin executing before s_i
  - if d$_i \neq \emptyset$, for d$_i$ =j:
    * if t$_j = \emptyset$, i does not execute
    * if t$_j \neq \emptyset$, i does not begin execution until after t$_j$

The utility of a schedule is defined as:
Utility(S) = $\sum_{i=1}^{n}$ util(task$_i$)
where

$$\text{util(task}_i) = \begin{cases} u_i(t_i), t_i \neq \emptyset \\ 0, \text{else} \end{cases}$$

The optimization problem, then, is to determine a schedule S such that Utility(S) is maximized. That is, find a schedule S such that:
legal(S) $\wedge$ ($\forall$ S') [legal(S') $\rightarrow$ Utility(S) $\geq$ Utility(S')]

Chapter 4

Program Description

This scheduler is implemented as an extension to an existing scheduling program, TimeWiz by TimeSys Corporation [25]. TimeWiz already allows for such scheduling heuristics as priority-based scheduling or deadline-based scheduling, and is a suitable framework for extension into the utility domain. In this program, there are three main items representing components of real-time systems: *resources*, *events*, and *actions.*

Resources are the entities that are being scheduled. Resources could represent anything schedulable: a tool in a workplace, a machine or piece of equipment, or (in the case of this implementation) a processor. TimeWiz allows for a resource to have multiple uses, such as different parts of a computer (e.g., memory, bus) which are controlled by a larger resource, such as a processor. For the purposes of this initial program, however, the problem being considered includes only a single processor as a single resource and does not require or utilize the additional functionality. In addition, while there are many additional properties of the processor available within the TimeWiz framework, for the purposes of this implementation the processor is considered to be a single "black box" resource and the inner workings of the processor are ignored.

Events are simply triggers for the purposes of this implementation. Each event has a start time, which corresponds to some point in simulated time. When this start time is reached the event is triggered, which means that the action dependent on it is introduced into the simulation. This time is known as the "release time" of the action. Note that for a series of dependent actions, the completion of one action in the chain acts as the trigger for the next action in the chain.

Actions are the items that actually utilize the resource. They can represent the jobs, tasks, or processes *submitted* to the processor. When an action needs to be executed on a processor, it is submitted to a processor to notify the processor that this action needs to run on it. Each action has a variety of parameters that describe its activity within the system. They each have their own utility functions, start times, execution times, and dependencies that allow for the actions to effectively model the system under study.

4.1 Utility Functions

There is no limit to the number of functions that can be utility functions; any one-to-one function is possible. In practice it is often sufficient to consider only a small subset of functions, which can be described fairly simply. This implementation only considers a set of functions which are broken up into two contiguous time intervals, so utility-wise they are defined from time t_1 to time t_2, and from t_2 to time t_3; everywhere else they are zero. Within each time interval they can be described by a quadratic equation, a linear equation, or a constant.

For this implementation, the quadratic and linear equations are somewhat different than the standard method of describing quadratic and linear equations. For linear equations, $y = mx + b$ is the standard equation. This implementation considers y to be the utility function u_i and x to be time, so the formulation could be rewritten as $u_i(t) = mt + b$. It is useful, in the case of this particular implementation, to consider the value of the functions at the three key time points of the function. Therefore, this implementation asks for the values of the function at t_1 and t_2 and the slope of the line and extracts the standard form equation from there. So, given:

- interval start time (t_1 or t_2)

- the slope m

- the utility at the start of the interval (h_1 at t_1 or h_2 at t_2, respectively)

it follows that:

$y(t_1) = mt_1 + b = h_1 \Rightarrow b = h_1 - mt_1$

Similarly, for quadratic equations, given:

- coefficients a and b
- interval start time (t_1 or t_2)
- the utility at the start of the interval (h_1 at t_1 or h_2 at t_2, respectively)

it follows that:

$$\begin{aligned} y &= ax^2 + bx + c \\ y(t) &= at^2 + bt + c \\ y(t_1) = at_1^2 + bt_1 + c &= h_1 \Rightarrow \\ c &= h_1 - at_1^2 - bt_1 \end{aligned}$$

4.2 Implementation Details

In any implementation, there are certain choices that must be made to represent the problem being approached. In the section on utility functions (Section 4.1), some details related to utility functions were described. There is one additional utility function detail: the code is designed to allow for easy addition of utility "subfunctions" (e.g., constant, linear, or quadratic) without serious *code blowup* (where code blowup is a large increase in the number of lines of code necessary to make the change), as long as they can be described with two coefficients and a constant start value. Besides adding code for the function itself, little code must be modified. In addition, there is not a great deal of difficulty in adding in new coefficients. However, it is a problem to add in a greater number of subintervals. There are multiple functions that deal with examining the subintervals, and code blowup can occur with increasing the number of subintervals. For example, suppose the number of subintervals is increased from two to three. To implement this, it is necessary to alter many different sections in the code by adding a new case to account for examining one more subinterval.

There are a few issues with processor scheduling which should be clarified for this particular implementation. For one, it is necessary to consider what the *setup time* is for each task. The setup time is the amount of time it takes to begin a task; in processors, this refers to such things as process creatin times and context switch times. For the purposes of this implementation, all setup times are set to zero. This is largely for simplification of the problem. If necessary to consider the setup time, it could be treated as either a dependency or by adding additional execution time to the action to account for the setup time.

Another issue that had to be addressed was the handling of multiple simultaneous release times. In other words, suppose five different tasks become available to the processor simultaneously. The question becomes whether to wait until all tasks at that time have become known to the processor before scheduling them or to immediately begin executing those jobs and simply keep preempting as additional tasks with the same release time arrive. In this implementation the latter is utilized, even though the former is what appears most often in reality. Take the case of the five tasks becoming available at the same time. This acts like a list of the five tasks becoming available. This implementation starts executing the first task on the list, then looks at the second to possibly preempt the first, and so on. The choice is purely based on ease of implementation; however, since for this project all setup times are defined to be 0, there is no harm in utilizing the multiple preempt approach.

4.3 Code Structure

This project is implemented as a plug-in to TimeWiz [25]. TimeWiz is a Windows©application designed for extensibility to other scheduling and analysis disciplines. This extension is implemented by compiling a dynamic link library file written in C++, UtilityAnalysis.dll in this case. The details of setting up a plug-in to TimeWiz are discussed in Appendix A.

There are a number of functions that may be called by the plug-in to interact with the TimeWiz interface. This implementation uses four of these functions in a way that is important for proper execution of the utility-based real-time scheduling: three for simulation, one for analysis. In addition, each of these functions calls other functions written specifically for this implementation.

Two of the scheduling functions used in this implementation are *StartSimulation* and *EndSimulation.* These functions are called at the beginning and the end of a run of a simulation, respectively. For this implementation, these functions are used to initialize and clear variables, specifically initializing dependencies and initializing the choice of scheduling policy used for that particular simulation.

The most important scheduling function is *SimulateAction.* The overall simulation is run with the TimeWiz simulation engine. The TimeWiz engine keeps track of the running time in simulation, what is executing on the processor, and the structure and relevant times of the actions and events being scheduled. When an action is triggered, either by an event or completion of another action it is dependent on, the simulation engine makes a call to *SimulateAction.* This call initiates a *SimulateAction* thread specific to that action. Thus, each action has a *SimulateAction* thread directly associated with it.

This implementation uses *SimulateAction* as a means for managing task execution in a distributed fashion. UtilityWiz keeps track of a global queue where all actions that have been triggered and are either executing or waiting to execute reside. Every simulated time where an action is triggered, it is added to the queue in the appropriate location based on some measure of utility, possibly preempting the currently executing action. Once an action completes execution, the queue of actions is reorganized based on newly assessed utilities for all actions that remain. This continues until the end of simulation.

SimulateAction calls a set of functions in this implementation that are specific for utility-based scheduling. When an action is first triggered, the *SimulateAction* thread created for that action calls *ReviseSched.* This function initially places the action in the proper place in the execution queue by comparing the value obtained by *GetDeadlineVal* to the other values in the queue, and preempts the currently running action if necessary. When an action completes, the queue is revised by a call to

ReviseQueue.

The analysis function that interacts with the TimeWiz interface is the *Calc* function. This function is called by TimeWiz when the "Run Analysis..." option is selected by the user. This implementation uses the *Calc* function to calculate three values: the utility accrued in the most recent simulation run, an estimate of the maximum accruable utility for this set of tasks (across all possible schedules), and the total runtime of the *Calc* function in milliseconds.

Calc utilizes a set of analysis functions written for this implementation. *Calc* initializes some of the conditions specific to that particular analysis run, then calls *GetMaxUtilWithSubInt.* This function initializes variables and arrays specific to the analysis, and then calls *divideUp. divideUp* runs through the sets of actions, and for each set of actions *findValid* is called. *findValid* calls *RunThroughCombinations*, which runs through the different permutations of each set of actions. *DependenciesOkay* and *viable* are called from there to determine that all of the constraints are met.

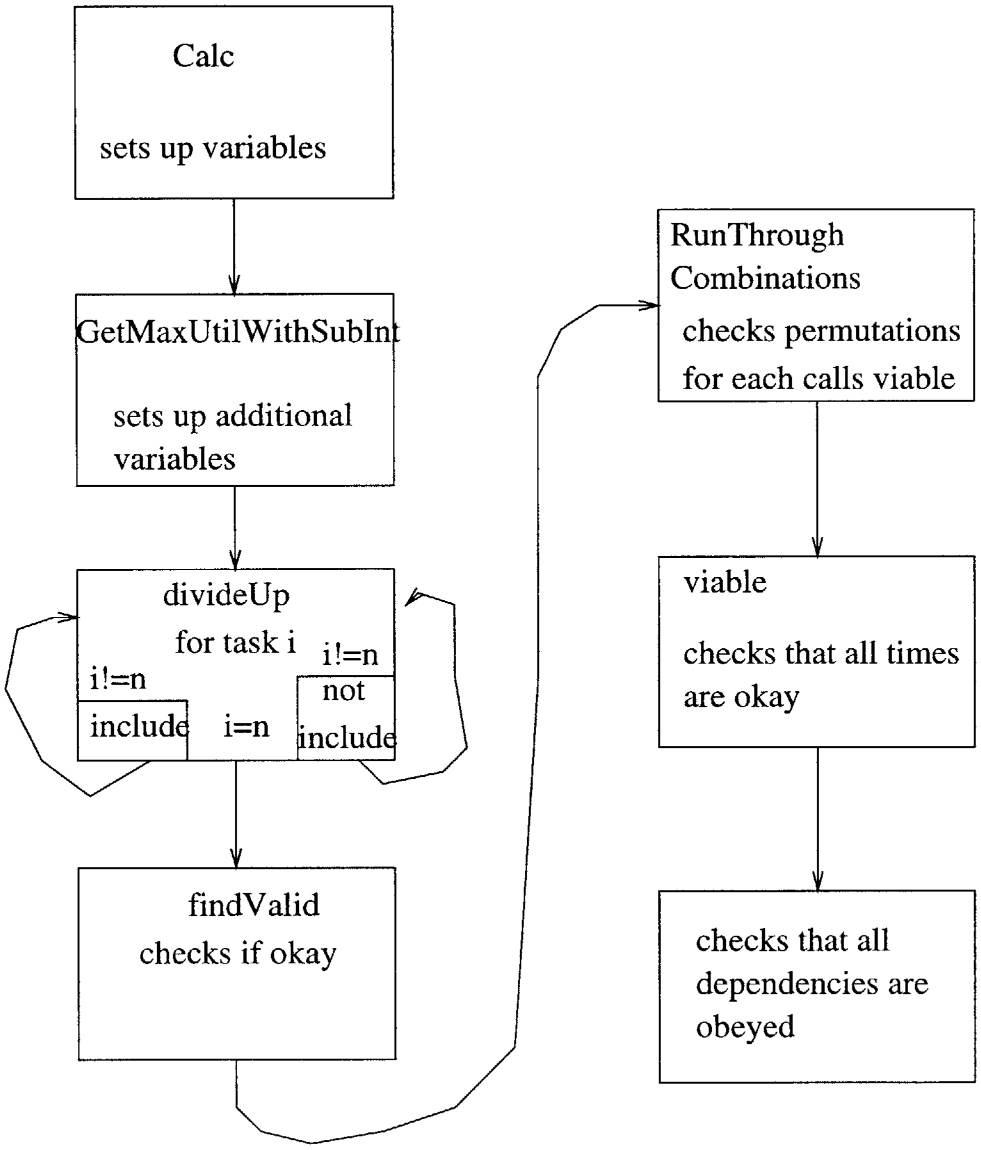

Figure 4-1: Flowchart of Analysis Code

Chapter 5

Implemented Scheduling Policies and Analysis

5.1 Scheduling Policies

There are two scheduling policy options available in this particular implementation. The first one is EDF-var, as discussed in section 2.1.1. This option is implemented more as a matter of comparison than anything else, since it does not explicitly consider utility. The second option is designed for utility-based scheduling. This option, called GreedyUtil, is a greedy scheduler, implementing a scheduling approach that is based on what is best to run at any particular point in time.

GreedyUtil keeps a running queue of all actions that are ready to run. Any time a new action is triggered and is added to the queue or an action finishes and leaves the queue, the scheduler rearranges the queue. It looks at the total execution time remaining for each action in the queue. It then evaluates the utility of each action at the time it would finish were it to run to completion without preemption; that is, at the time t which is equal to the current time in simulation plus the time remaining for the process to execute in the simulation. If the action cannot complete by its deadline, it is removed from the queue. The scheduler takes the value of completing the action at time t and divides it by the time it takes to complete the action, getting the utility per time unit. The queue is then reordered by decreasing utility per time unit; thus, while the next action scheduled may not give the highest utility at completion, it will get the best utility per time unit.

More formally, considering

- a set of tasks 1,...,n
- $\forall$ i, e(i) = execution time remaining on task i:
- $\forall$ i, $u_i(t)$ = utility of task i completing at time t

GreedyUtil

```
Main:
Time t ← 0
Queue q ← NULL
While t < T (= total simulation time as set in code)
    If $|q| > 0$
        execute 1st element in queue until one of the following is true:
            it completes
            a new task is released
            current time = T
        If 1st element completed
            remove from queue
            ReorgQueue()
            t ← current time
    Else
        wait until a task is released or end of simulation is reached
        t ← current time

ReorgQueue:
For i' ← $|q|$ down to 1
    If t > deadline(i) (where deadline = time $t_3$ from Section 4.1)
        Remove i from queue (where i = q[i'])
    Else
        For j' ← i' to $|q|$
            if $\frac{u_i(t+e(i))}{e(i)} > \frac{u_j(t+e(j))}{e(j)}$
                insert i in front of j (j = q[j'])
                break
```

This scheduler will often not obtain the optimal utility-based schedule, as it only looks at the local best rather than the best for all time. So, for example, suppose the utility function of some action were simply a straight line with a slope of one starting at time 0 and ending at the end of the simulation. The best time for this action to finish would be as near the end as possible; however, the greedy scheduler may schedule it first, obtaining a far from optimal value.

5.2 Analysis

The problem of finding the optimal schedule in the case of utility-based scheduling is actually an NP-hard problem[11]. Therefore, to analyze the level of success of a given schedule, it is much more tractable to estimate the optimal rather than calculate it exactly. The analysis portion of the tool gives an approximation of optimal; the longer taken by the approximation, the closer to optimal the result.

5.2.1 Algorithm

The algorithm takes the continuous utility functions and breaks them into discrete chunks that assume the maximum value in that chunk. For example, suppose the total execution time of the simulation is 100 time units and the number of divisions chosen is ten. The approximation will create a new piecewise step function for each action which has constant value from 0-10 time units, a different value from 10-20, and so on. The constant value it assumes is the maximum value the original utility function assumes for the interval. An example of the approximation process is shown in Figures 5-1, 5-2, and 5-3.

Once the function has been discretized as described above, the algorithm implements something similar to a branch and bound algorithm to search the solution space [5]. First, consider the number of possible solutions. Suppose, for some run of the algorithm, there are n actions and a total of m different time intervals. Consider an artificial (m+1)st time interval, where all actions that are not completed during the execution time of the simulation are considered to complete in this time interval.

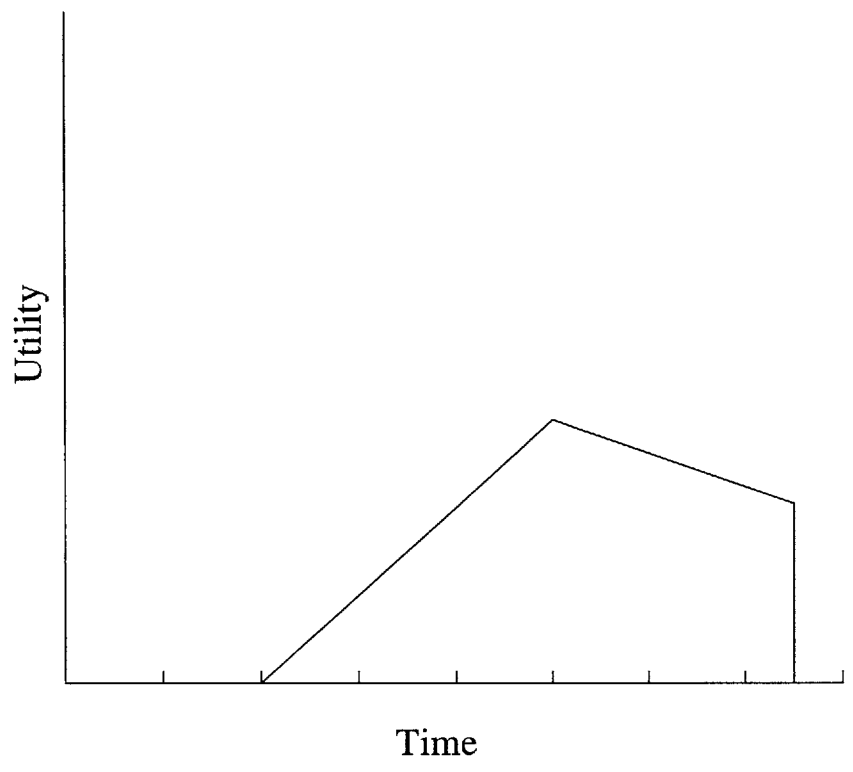

Figure 5-1: Original utility function

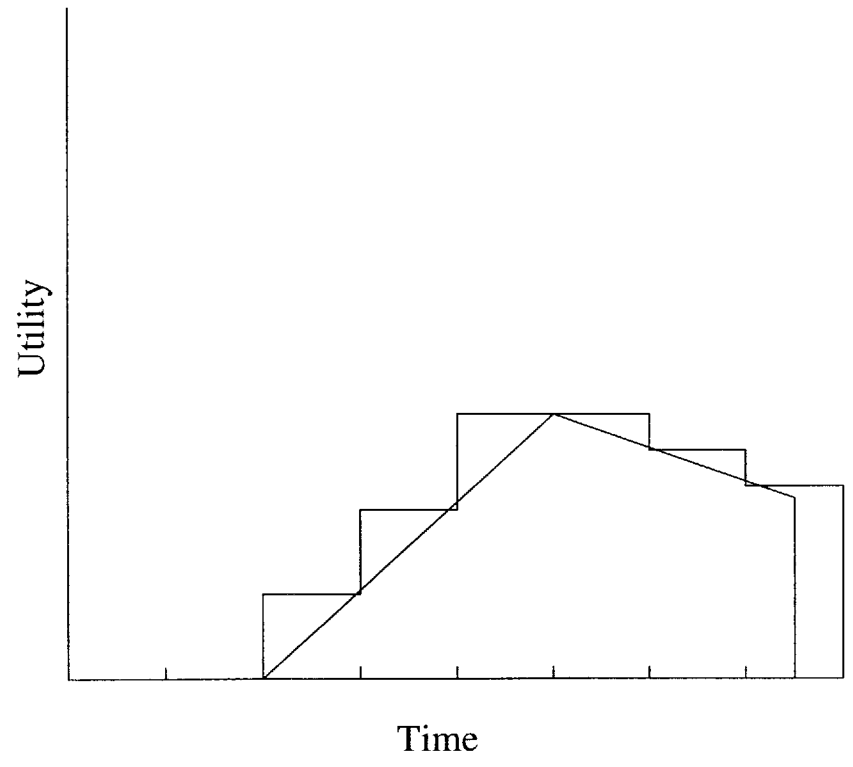

Figure 5-2: Approximation superimposed on utility function

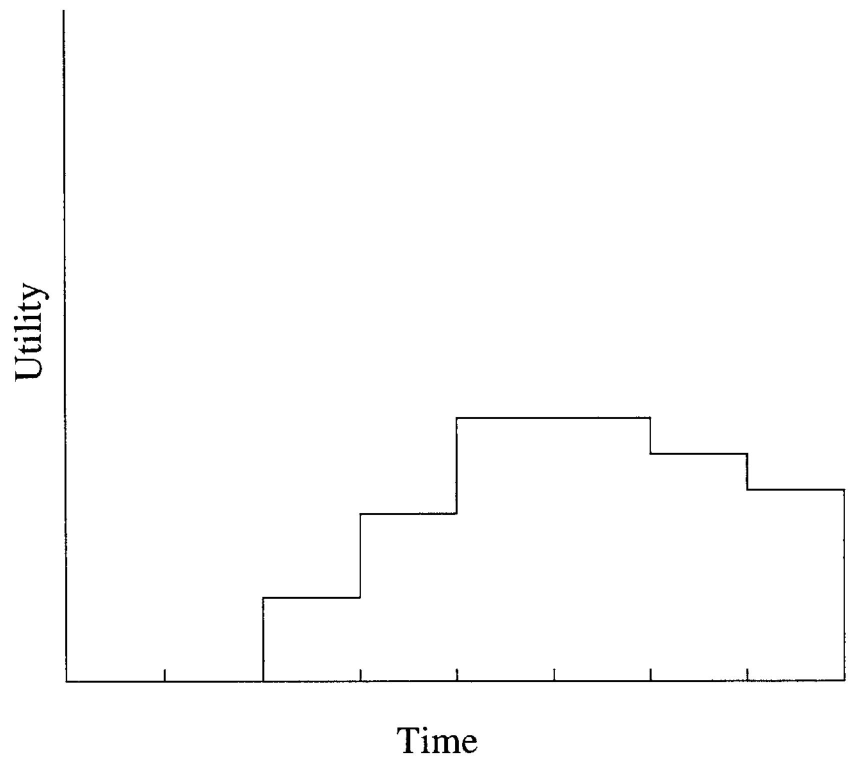

Figure 5-3: Approximation alone

Every action that completes ends in some time interval, and it ends with a utility less than or equal to the maximum utility in that interval. In fact, if it is assumed that both subintervals of a utility function are monotonically increasing, monotonically decreasing, or constant, then the maximum in each of the m intervals is either at an endpoint of the interval or at a *key point* (beginning or end of of one of the subintervals of the utility function). Thus, the optimal solution of the utility-based scheduling problem is less than or equal to the optimal solution of the utility-based scheduling problem with this new step function. Furthermore, a lower bound can be obtained[1] by taking the minimum utility of each task in each of the m intervals and using that for the step function. Maximizing this value would give a lower bound on the value of the optimal solution. These two approximations sqeeze the actual value, as seen in Figure 5-4.

The number of possible solutions to this problem is still large, but an important

[1]Not implemented in this module

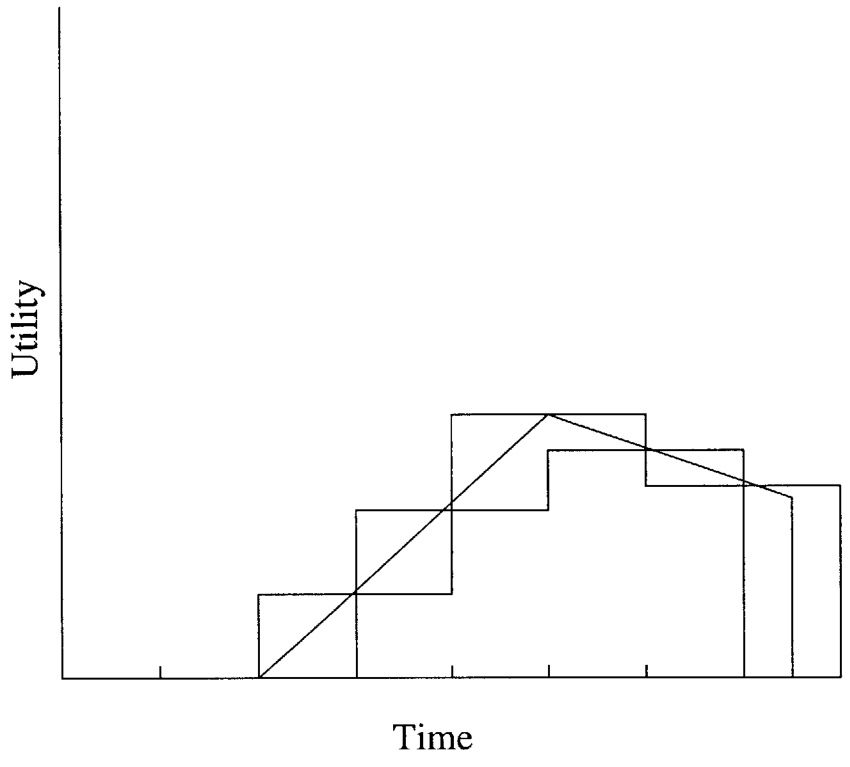

Figure 5-4: Lower and Upper Approximationd Superimposed on Original Function

observation simplifies this: the new piecewise utility function has the same value for the full length of every interval. Thus, it is sufficient to find actions that complete by the end of the interval and not worry about exactly when they complete within the time interval. The problem can then be considered as placing actions in "buckets" based on time of completion. There are m+1 different buckets, so each action has m+1 different possible completion time slots. Thus, there are $(\mathrm{m}+1)^n$ different possible combinations for placing the n actions into m + 1 buckets. In addition, any of these schedules can be checked in O(n) time. Therefore, the naïve approach of simply trying all possible time/bucket assignments and checking them for correctness takes $O(n(m+1)^n)$ time.

It is possible to improve greatly on the running time of the algorithm in many cases, although asymptotically the running time is the same. This is done by adding in optimizations while checking the viability of schedules to shortcut the number of schedules checked.

First, consider the manner in which schedules are checked. To allow for greater flexibility in optimization, consider a scheme in which all possibilities are checked recursively through the actions. That is, for each of the N tasks i $\in$ {1,...,n}, set action i to execute and check all possibilities for actions (i+1) to n. Subsequently set action i to not execute and check all possibilities for actions (i+1) to n. Finally, take the maximum of these two results. If this approach is considered, it is then possible to consider some optimizations based on the recursion in constructing the different potential schedules.

One of these optimizations is limiting the number of possibilities checked by not adding additional items to a schedule if no more can be added such that the schedule remains feasible. This is implemented by setting up an array at the beginning of the analysis that tells the minimum length execution time of any of the remaining tasks. This is done by looking at each action starting at the end of the list of actions (there is a list of actions maintained by the resource that can be accessed from within the code). Initialize the "min-time" array by setting the nth element of the array to be the execution time of the nth action. Note that the order is based on the order in the action array and is independent of actual order of execution. For any action i, look at action i+1 (the action previously examined). If the execution time of action i is less than the (i+1)st element of the "min-time" array, then the ith element of the "min-time" array is set to the execution time of element i. If not, the ith element of the "min-time" array is set to the (i+1)st element of the "min-time" array. What this produces is an array of the minimum execution times remaining.

With this array, it is possible to look at any point in the recursion and check if it might be feasible to add an additional action and not exceed total running time. If not, simply set the remaining actions to not execute. This simple "pruning" gives the ability to only look at the combinations of schedules that might execute based on total running time alone. It eliminates the finding of permutations of schedules that cannot be met under any circumstances simply due to total execution time. In addition, this approach does not even examine a subset of the variables that cannot be added in.

Another simple optimization that greatly reduces the runtime of the analysis in the average case is to keep track of the best result so far. Before checking the different possible permutations of the schedule, find the maximum utility value of each utility function that will be considered in this particular combination of tasks and sum them up. If the sum is not greater than the maximum value achieved so far, then there is no need to check the possible permutations of this set of tasks, as it is not possible to improve on the current best. This "pruning" on average greatly reduces the number of permutations examined, and the permutations take up a large amount of total running time in the analysis.

Based on these optimizations, the algorithm is as follows:

Division-analyze

```
Main:
Initialize "min-time" array, max so far ( = 0), all tasks (to not executing)
Return ZeroOne called with current task equal to 0


ZeroOne(current task):
If over total time in simulation (as defined in code)
    return 0
If no tasks can be added without exceeding total time in simulation
    set all remaining tasks to not execute
    return CheckAll()
If at last action
    return CheckAll()
Else
    Set current task i to executing
    Call ZeroOne(current task = i+1)
    Set current task i to not executing
    Call ZeroOne(current task = i+1)
    Return the max of these two ZeroOne calls


CheckAll():
if max possible for this set of actions is greater than max so far
    go through all permutations
    return the max utility permutation that satisfies temporal and
        precedence constraints
if not
    return max so far
```

5.2.2 Analysis of Algorithm

Each of the three parts of Division-analyze has a purpose in the algorithm. *Main* sets up some necessary data. *ZeroOne* runs through all of the possible sets of actions. In *ZeroOne*, one of the optimizations is performed. Combinations of tasks that cannot complete by the end of the simulation are eliminated without further examination and, in many cases, not even considered at all. *CheckAll* first checks if the sum of the maximum utilities of all tasks in the current combination is greater than the maximum total utility achieved so far; if not, there is no need to check further. If so, it runs through all possible permutations and takes the maximum permutation that meets all scheduling constraints. *CheckAll* performs another optimization, by only checking tasks in the intervals where the utility function is defined. In this fashion, all of the possible ways of legally placing the tasks into the intervals are checked by Division-analyze.

The algorithm will always give a set of time slots that is feasible. When the algorithm actually assigns the actions to the time slots, the schedule is feasible. However, the values of the approximated utility function (see Figure 5-3) that are reached may not be possible. In fact, the difference between the value of the optimal solution possible and the solution presented by the algorithm is less than or equal to the sum of the differences between the maximum utility and the minimum utility of each action in the interval where each action is assigned in this solution. Implementing the approximation with the minimum value in the intervals (as described in Section 5.2.1) will give a lower bound for the maximum utility, so the maximum utility is between the upper bound obtained in this implementation and the lower bound as described in Section 5.2.1.

Chapter 6

Description of Variables

Each component in TimeWiz has a set of variables specific to that component. The variables most relevant to the UtilityWiz plug-in are listed below.

One implementation detail to note is that when the situation arose in which a choice had to be made between different options (e.g., choice of scheduling policy) the choice was always between integer values. This choice was made due to the greater simplicity of working with integer values in TimeWiz.

Table 6.1: Resource-Specific Input variables

Variable name in code	Title in interface	Values assumed	Description
TWZName	Name	String	Name of resource, default is CPU
TWZScheduling Heuristic	Scheduling Heuristic (100=EDF, 101=GreedyUtil)	100, 101	Scheduling policy used when a simulation is run; 100 means EDF-var, 101 means greedy utility
TWZSubdivisions	Subdivisions	positive Integers	Number of segments or chunks of time that the total time in simulation is divided into for the purpose of analysis

Table 6.2: Resource-Specific Output variables

Variable name in code	Title in interface	Values assumed	Description
TWZUtility	Utility	non-negative float	Total accrued utility of all actions reached in most recent run of simulation of all the actions; calculated when analysis is run
TWZUtilityPossible	UtilityPossible	non-negative float	Upper bound on the maximum achievable accrued utility; calculated when analysis is run
TWZTimeElapsed	TimeElapsed (ms)	Non-negative int	Total runtime of analysis in milliseconds; calculated when analysis is run

Table 6.3: Event-Specific Input Variables

Variable name in code	Title in interface	Values assumed	Description
TWZName	Name	String	Name of the event
TWZSimFirst Arrival	first arrival in simulation	Float	Time that the event occurs in the simulation and triggers its associated action

Table 6.4: Event-Specific Output Variables

Variable name in code	Title in interface	Values assumed	Description
TWZResponse	Response	String	Ordered list of action(s) dependent on this event; for example, "act->act #2" means act #2 depends on act which is triggered by this event

Table 6.5: Action-Specific Input Variables

Variable name in code	Title in interface	Values assumed	Description
TWZName	Name	String	Name of the action
TWZTime PointOne	TimePoint One	Non-negative Float	First time point at which the utility function is defined; utility = 0 for all earlier times
TWZStart ValOne	StartValOne	Float	Value the utility function assumes at time point one
TWZFirst ParameterOne	First ParameterOne	Float	First parameter (if applicable) of the function describing the utility function from time point one to time point two
TWZFirst ParameterTwo	First ParameterTwo	Float	Second parameter (if applicable) of the function describing the utility function from time point one to time point two
TWZUtility FunctionOne	Utility FunctionOne	0,1,2	Description of the utility function from time point one to time point two; 0 means constant, 1 means linear, 2 means quadratic
TWZTime PointTwo	TimePoint Two	Non-negative Float	Time point at which the utility function changes from function one to function two
TWZStartVal Two	StartValTwo	Float	Value the utility function assumes at time point two
TWZSecond ParameterOne	Second ParameterOne	Float	First parameter (if applicable) of the function describing the utility function from time point two to time point three
TWZSecond ParameterTwo	Second ParameterTwo	Float	Second parameter (if applicable) of the function describing the utility function from time point two to time point three
TWZUtility FunctionTwo	Utility FunctionTwo	0,1,2	Description of the utility function from time point two to time point three; 0 means constant, 1 means linear, 2 means quadratic
TWZTime PointThree	TimePoint Three	Non-negative Float	Last time point at which the utility function is defined; utility = 0 for all later times

Table 6.6: Action-Specific Output Variables

Variable name in code	Title in interface	Values assumed	Description
TWZTime Remaining	Time Remaining	Float	Remaining execution time; only valid during a simulation run
TWZUtility	Utility	Float	Utility obtained by the action at time of completion during a simulation run
TWZDependsOn	DependsOn	String	Name of the action that depends on this action, if there is one
TWZIs DependentOn	IsDependentOn	String	Name of the action that this action is dependent on, if there is one
TWZPredecessor Complete	Predecessor Complete	0,1	1 if predecessor has completed, 0 if has not completed (ignored if there is no predecessor); only valid during a simulation run

Chapter 7

Results

There are two components in the UtilityWiz module, the scheduler and the analysis tool.

7.1 Scheduling Results

The greedy utility scheduler is known to be non-optimal, but one important question is whether it will run with appropriate data sets and how good the results will be. For this purpose, each of the following data sets was scheduled with both the EDF-var and the GreedyUtil policy. The results are as detailed below.

7.1.1 Step Utility Functions

For the simple step functions, there are two issues examined in the data sets: preemption/no preemption and increasing/decreasing value in step. To cover these issues, one case of each combination was examined in the simulator as described in Table 7.1.

Table 7.1: Step Utility Function Data Sets

	Action	Start time	Execution time	Time point one	Value in interval one	Time point two	Value in interval two	Time point three
St_1:	Act#1	0	100	0	30	50	55	150
	Act#2	0	100	0	60	110	45	200
St_2:	Act#1	0	100	0	60	50	45	150
	Act#2	0	100	0	30	110	55	200
St_3:	Act#1	0	100	0	30	50	55	150
	Act#2	50	100	0	60	110	45	200
St_4:	Act#1	0	100	0	60	50	45	150
	Act#2	50	100	0	30	110	55	200

7.1.2 Step and Linear Utility Functions

The linear and quadratic functions add in a new dimension to utility functions. For these functions, the value is increasing or decreasing for a segment, so it makes a difference where the function is evaluated. To test this property, it is sufficient to test for only the performance on the linear functions, based on the previous assumption that all utility functions are monotonically increasing, monotonically decreasing, or constant within each subinterval. In the data set described by Table 7.2, all non-step functions composed of an increase/decrease/constant pair are compared against a step function they cross.

Table 7.2: Step and Linear Utility Function Data Sets

	Action	Slope in segment 1	Slope in segment 2	Time point one	Value in interval one	Time point two	Value in interval two	Time point three
Li_1:	Act#1	0	0	0	140	110	160	150
	Act#2	0	1	0	100	100	100	200
Li_2:	Act#1	0	0	0	140	110	160	150
	Act#2	0	-1	0	200	100	200	200
Li_3:	Act#1	0	0	0	140	110	160	150
	Act#2	1	1	0	100	100	200	200
Li_4:	Act#1	0	0	0	140	110	160	150
	Act#2	1	-1	0	100	100	200	200
Li_5:	Act#1	0	0	0	140	110	160	150
	Act#2	1	0	0	100	100	200	200
Li_6:	Act#1	0	0	0	140	110	160	150
	Act#2	-1	1	0	200	100	100	200
Li_7:	Act#1	0	0	0	140	110	160	150
	Act#2	-1	-1	0	300	100	200	200
Li_8:	Act#1	0	0	0	140	110	160	150
	Act#2	-1	0	0	200	100	100	200

Table 7.3: Step and Linear Scheduling Results

Data set	EDF-var Act#1 utility	GreedyUtil Act#1 utility	EDF-var Act#2 utility	GreedyUtil Act#2 utility	EDF-var accrued utility	GreedyUtil accrued utility
St_1	55	0	45	60	100	60
St_2	45	45	55	55	100	100
St_3	55	55	45	45	100	100
St_4	45	0	55	55	100	55
Li_1	140	140	200	200	340	340
Li_2	150	0	200	200	350	200
Li_3	140	0	100	200	240	200
Li_4	140	0	100	200	240	200
Li_5	140	0	200	200	340	200
Li_6	140	140	200	200	340	340
Li_7	140	0	100	200	240	200
Li_8	140	140	100	100	240	240

7.1.3 Summary of Scheduling Results

Table 7.3 compares the results obtained for both EDF-var and the greedy utility scheduler.

The EDF-var outperformed the GreedyUtil algorithm in many cases. The reason for this is simple: the GreedyUtil algorithm is designed for the case of much system overload. In this case, the disadvantage that the greedy algorithm has of possibly executing something that will block some higher total utility functions from executing is greatly decreased, as it is much less probable that there will be processor down time. In this case, since GreedyUtil optimizes utility per unit time, GreedyUtil performs much more strongly.

7.2 Analysis Results

The level of success of the analysis portion of the tool can be described as the closeness of approximation to optimal and the amount of running time it took to reach it. One set of test runs to evaluate the success of the analysis tool was performed with the set

Table 7.4: Actions for Analysis Data Set

Action	Start time	Execution time	Time point one	Value in interval one	Time point two	Value in interval two	Time point three
Act	0	100	0	0	90	50	100
Act#2	0	100	0	0	110	30	200
Act#3	0	50	0	10	150	20	300
Act#4	0	50	0	20	200	30	300
Act#5	20	20	0	60	30	50	300
Act#6	20	40	20	50	30	40	60
Act#7	100	20	50	10	100	70	400
Act#8	300	100	100	100	150	20	400

of actions described in Table 7.4. Note that all of these functions are step functions; this choice was made to allow for calculation by hand of the maximum achievable utilities.

The first simulation, denoted Act_2, was composed of the first two actions in Table 7.4. There were a series of seven simulations, denoted Act_2 to Act_8, where Act_n is composed of the first n actions in the above table. Each simulation was analyzed for increasing numbers of divisions of the total time in simulation. The results are displayed in Table 7.5.

For each simulation Act_2,...,Act_8, there exists a schedule that maximizes the possible accrued value. These were calculated by hand and are shown in the Gantt chart depicted in figure 7-1.

The first interesting schedule to notice is that for Act_5. In Act_5, it is necessary to shed task Act#3 to achieve maximum accrued utility. In fact, it is impossible to schedule these tasks without shedding at least one. The second interesting schedule to notice is that for Act_8. In Act_8, it is impossible to feasibly achieve maximum accrued utility without preempting a task; in this case, Act#3 is preempted.

Each of the systems achieved some accrued value in simulation. Table 7.6 compares the accrued value achieved in simulation, the maximum achievable accrued utility, and the estimate on maximum utility achieved in each run of the analysis

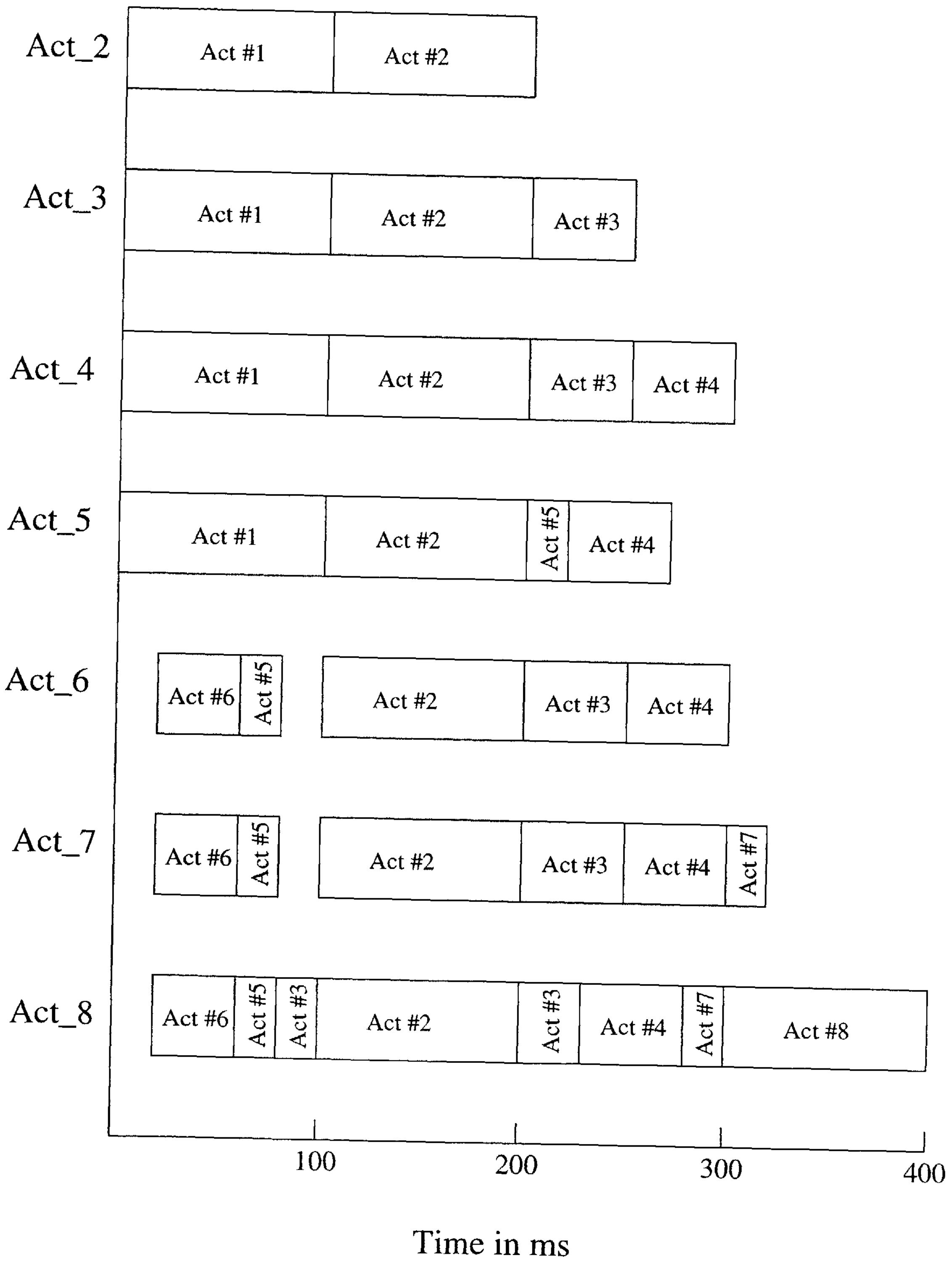

Figure 7-1: Gantt chart for systems Act_2 to Act_8

Table 7.5: Runtime of Analysis (ms); * means did not complete

#divs	Act_2	Act_3	Act_4	Act_5	Act_6	Act_7	Act_8
5	1	3	8	19	43	93	267
10	1	4	10	29	81	173	1209
15	2	4	13	53	327	1402	16460
20	2	6	21	125	914	5524	*
25	3	7	39	349	2339	37454	*
30	3	9	58	669	6887	*	*
40	3	13	128	2237	*	*	*
50	4	24	359	16280	*	*	*
60	5	38	822	*	*	*	*
70	7	58	1586	*	*	*	*
80	7	83	3482	*	*	*	*
90	9	139	6841	*	*	*	*
100	11	189	27027	*	*	*	*

function.

Unfortunately, as can be seen in table 7.5, the exponential factor quickly became too large for the system to handle even with the pruning of unneccessary info early on in the search. Although the size of the problem handled could be improve with a faster computer with more memory, it would soon be outstripped by the exponential growth in operating time.

7.3 Implementation Difficulties

There are some problems with the current implementation, which will need to be corrected for full functionality of the tool. First, there is some difficulty in extracting the dependency relationships of the various actions from within the plug-in. The current implementation therefore does not consider dependencies at all in the analysis portion, leading to the possibility of a large overestimate. In addition, there are some difficulties that arise with the simulation of dependencies as well. If task i is dependent on task j, the current implementation will consider the release time of task i as the release time of task j plus the execution time of task j. Thus, if task j is preempted,

Table 7.6: Analysis of Utility

#divs	Act_2	Act_3	Act_4	Act_5	Act_6	Act_7	Act_8
Utility accrued in sim. (EDF)	80	100	130	130	170	240	260
Utility accrued in sim. (greedy)	80	70	90	120	120	160	180
Max possible utility	80	100	130	160	170	240	260
5	80	100	130	190	220	290	340
10	80	100	130	190	210	280	330
15	80	100	130	180	180	250	320
20	80	100	130	180	180	250	*
25	80	100	130	180	180	250	*
30	80	100	130	180	180	*	*
40	80	100	130	180	*	*	*
50	80	100	130	180	*	*	*
60	80	100	130	*	*	*	*
70	80	100	130	*	*	*	*
80	80	100	130	*	*	*	*
90	80	100	130	*	*	*	*
100	80	100	130	*	*	*	*

the simulation behaviour can be incorrect.

The functionality is fully in place once the dependency relationships can be established. In addition, there has been some limited testing in which the dependency relationships for one particular set of dependent actions were entered manually into the code. In this testing, all dependencies were obeyed, both in the analysis and in the simulation.

Most of the other problems arise with difficulties interfacing the plug-in with the full range of TimeWiz's functionality. For one, there is an error message that frequently appears during most simulation runs[1]. Clicking Ignore on this error message every time it appears will eventually lead to obtaining the desired results, so the error is no more than an annoyance. Another difficulty is the inability to produce a correct Gantt chart. A Gantt chart is a graph that models machine use vs. time, where whatever action is using the machine is represented by a unique numbered or colored bar [4]. The Gantt chart is a very effective tool for visual representation of a schedule, and is implemented in TimeWiz through the "timeline" function. Unfortunately, the plug-in has not been able to produce the correct Gantt chart for its schedule. Finally, there are a set of output windows at the bottom of the interface which allow for text to be displayed to the user, which this implementation has not been able to access to output implementation-specific text.

One remaining problem is actually a problem that depends on the capabilities of the computer on which TimeWiz is executing. The Time Elapsed (ms) variable is set by making two calls to the system counter, subtracting the first value from the second, dividing by the frequency of the counter, then multiplying by 10^3 to get milliseconds elapsed. The difficulty with this is that not all computers are able to handle the 64-bit value used for the system counter and the frequency. For these machines, the Time Elapsed value will always be zero.

[1]Error: "The value of ESP was not properly saved across a function call. This is usually a result of calling a function declared with one calling convention with a function pointer declared with a different calling convention."

Chapter 8

Future Work

There are some unexplored possibilities and directions that hopefully will be fruitful for future work.

8.1 Linear Programming Analysis

In 1959, H.M.Wagner proposed an integer programming formulation of a general version of the processor scheduling problem. This formulation considered a single processor and immediate ready time for all jobs. In addition, this formula assumed all jobs would finish, and had some notion of idle time and wait time for processors. This formulation considered the permutations of running orders for the different processes with respect to the set of idle and wait times [11], [24].

The set of parameters for an integer programming formulation of the utility-based scheduling problem is somewhat different from this formulation. To describe the formulation, begin with the model where all execution, start, and simulation times are integral (the reasoning will be elaborated on later). If this is the case, all possible maximum solutions are integral as well. Begin by dividing the total running time T in simulation into J-1 different unit length time chunks (where J-1 = integral total execution time in the simulation). In addition, consider a Jth time chunk which means that all actions that complete in that time chunk do not complete in simulation. For each process i, assign a cost c_{ij} where c_{ij} is the utility value of completing activity i at time j. In addition, let x_{ij} be 1 if action i completes in time unit j. This gives an objective function of

$$\text{Maximize} \quad \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij}$$

subject to:

$$x_{ij} \in 0, 1$$

There are some additional necessary constraints. First of all, note that no task can finish before the sum of its start time and its total execution time. Therefore, add in the constraint:

$$x_{ij} = 0 \forall j < \text{start time + execution time of i}$$

In addition, the formulation must guarantee that the total execution time of all actions that have completed by each time point is less than the total time up to that point. This guarantees that the schedule is feasible with respect to execution time. Thus, add in another constraint:

$$\sum_{i \in I} x_{ij} * \text{execution time of process i} \leq \frac{(j+1)T}{J} \forall j \in J, j \neq J$$

The formulation also needs to guarantee that all dependencies hold, so dependent tasks do not begin executing until the tasks they are dependent on have completed. To guarantee this, add in the final constraint

$$x_{ij} + x_{i'j'} = x_{ij} \forall j' < j + \text{execution time of i'}, i' \text{dependent on} i$$

Considering all constraints together, the complete integer programming formulation is described as follows.

$$\text{Maximize} \quad \sum_i \sum_j c_{ij} x_{ij}$$

subject to:

$$x_{ij} + x_{i'j'} = x_{ij} \forall j' < j + \text{execution time of i', i' dependent on i}$$
$$\sum_{i \in I} x_{ij} * \text{execution time of process i} \leq \frac{(j+1)T}{J} \forall j \in J, j \neq J$$
$$\sum_{j \in J} x_{ij} = 1 \forall i$$
$$x_{ij} = 0 \forall j < \text{start time} + \text{execution time of i}$$
$$x_{ij} \in 0, 1$$

The difficulty for the integer program is that it is only correct for a certain subset of utility problems. The integer program requires, first of all, that every significant length of time be divisible by some factor. Significant, in this case, means earliest start time of each action, total time in simulation, time length of the first segment of each utility function, and time length of the second segment of each utility function. This is true if all values are integral, or even if all values are rational, but not for all reals. This creates a problem for trying to analyze a probabilistic distribution, which is one of the future objectives of this project.

In addition, the integer programming formulation is solvable in polynomial time, but it is polynomial in the number of actions and in the total length of time divided by the size of the minimal subdivision. For example, suppose the total length of time allowed for the schedule is 10^3 seconds. Suppose, however, that the greatest common divisor is 10^{-6}. Then number of time subsegments being looked at is of size 10^9. This can blow up very quickly with a negligible time difference so a scheduling problem with execution times of, say, 100.0000001 and 20 will take a long time but one with execution times 100 and 20 will not.

One more difficulty with this formulation is that it does not account for start times fully. The formulation guarantees that nothing will finish before its start time plus its run time. However, it does not consider the case where some processor time is not used because of start times. An example of a case which the integer program does not handle correctly is the case where two actions start at time 2, execute for 2 time units, and finish at time 4. In reality, it is not possible to schedule both, as only one may execute from time 2 to time 4, but the integer programming formulation does not recognize that.

The advantage of the integer program is that it can be relaxed to a linear programming formulation, which can be solved in polynomial time. In addition, if it is possible make the assumption that all relevant time units are integral, then assignment of the c_{ij} based on the value of the utility function of action i at each time point j will actually give the optimal solution with the integer program. Unfortunately, the integer program is not solvable in polynomial time, but if the constraint $x_{ij} \in \{0,1\}$ is relaxed to $x_{ij} \geq 0$, the integer program becomes a linear program that can be solved in polynomial time using such standard methods as the simplex method or the interior point method.

8.2 Introduction of Probabilistic Distributions

In many real-time systems, there are tasks which do have a known execution time, but rather have a probability distribution describing when an action is likely to occur or how long it is likely to take to finish execution. This would allow for a more realistic model of this sort of scheduling problem. The one large drawback is that it would complicate the analysis greatly. It would render an integer or linear program analysis infeasible, due to the lack of a specific time interval which divides into all elements of the problem. Some item that completes at an irrational time is not plausible for the integer and linear program representations.

At the same time, it complicates the analysis greatly. The question becomes what values the variables take in the analysis. Many options are available and reasonable for just the execution times alone. The analysis could analyze a particular case compared to the expected case, a particular case compared to how well it might have done given that certain set of execution times, the average execution times, the average execution times in some set of simulations, or the expected execution times just to name a few. Allowing for probabilistic distributions of the variables allows for greater opportunities but requires greater choices.

Chapter 9

Related Work

9.1 General

Real-time analysis can benefit a wide variety of real-world systems. These systems need to meet a variety of different needs, and as such have a wide variety of definitions of optimal in terms of schedules. The problem is known as the general job-shop problem. The general job-shop problem is defined as given n jobs that need to be executed on m machines, create a feasible schedule that optimizes some specific objective [11]. A feasible schedule is a schedule where all constraints specific to the problem are met.

There are numerous different real-world situations that can be described as job-shop problems. For example, one general job-shop scheduling problem is construction of a brick-built workshop. This problem has a certain number of skilled and unskilled labourers. In addition, there is a certain set of tasks that must be performed to build the workshop. These tasks have time and precedence constraints: building a wall or a frame takes a certain amount of time, and the roof can't be started until the walls are in place. The "machines" or resources are the labourers, and the tasks are the different jobs that must be done to complete construction of the workshop [21].

Another general scheduling problem is in the realm of air traffic control. Each flight needs to take off by a certain time from a certain airport, fly for a certain amount of time, and land in a certain airport. One problem that is encountered is congestion, namely the number of planes in the same airspace at the same time.

Consider each flight path as a set of points that a plane needs reach in order, where the first and last points are the takeoff and landing points, respectively. If proximity of these points is considered, then the scheduling problem can be to schedule the flights in such a way as to minimize the number of flights that are in close proximity while maintaining all precedence constraints (each flight must reach the set of points in a specified order, there is a certain amount of time needed for each flight to reach point a from point b) [6].

9.2 Utility-based

Utility-based schedulers have been implemented in a variety of different arenas up to this point. One implementation was in a US Air Force Advanced Technology Demonstration (ATD). This project is design of an adaptive distributed tracking system for the Airborne Warning and Control System (AWACS). Different tracks were given different value functions to describe their importance. The issue addressed by this implementation is how the AWACS scheduler performs under overload condition. This implementation met with marked success, as the tracks that were deemed "more important" (and thus given higher utility) were processed with a great deal of reliability even under overload conditions [7].

Another example of an implementation of a real-time utility-based scheduling policy is the Dependent Activity Scheduling (DASA) algorithm. The DASA algorithm is a utility-based algorithm that considers a smaller class of utility functions, namely those composed purely of step functions. This algorithm is particularly effective in a set of real-time systems known as supervisory control systems. It was simulated with very positive results, especially in the case of overload [8].

There have been multiple implementations of utility-based real-time scheduling in the realm of microkernels. A utility-based policy called the Best Effort policy was successfully implemented in the Alpha operating system developed at Carnegie Mellon University [22], [14]. In this environment, another tracking scenario was implemented under real-time scheduling policies to a high level of success [23]. The Best Effort scheduling policy was later adapted as one of the multiple policies implemented in the MK 7.3a microkernel [1].

In addition, there is currently a specification for real-time Java under development. The extension of the language under development is real-time CORBA 1.0. This specification includes provision for real-time utility-based scheduling, especially distributed scheduling [15], [3], [2].

A large chunk of real-time work ([25], [26]) to this point has used the Rate Monotonic Analysis approach. This approach as described above can guarantee that a system is schedulable, assuming a certain set of requirements is met [18]. There have also been implemented some utility-based schedulers in practice, with very positive results. Some utility based schedulers have already been implemented ([23], [1], [7]), although until now there has been no current analysis tools for this type of scheduling. These may also in the future be implemented in the specification for real-time Java [3].

Chapter 10

Conclusions

Utility-based real-time analysis is a view of scheduling that can be applied to a variety of real-world real-time systems [23], [7]. In addition, this approach can describe a variety of other scheduling approaches (such as EDF and priority-based) if utility functions are carefully chosen; if the utility function of each task is simply $\frac{1}{\text{deadline}}$, this models the EDF problem. The problem with this real-time paradigm is that it is hard to analyze how optimal any schedule is. The results from running the analysis algorithm with this module show that even a good approximation of optimal can be an exponential problem. For large cases, it is necessary with the current level of knowledge to make more limiting assumptions than were made for this implementation for this analysis to become tractable. Nonetheless, utility-based scheduling of systems remains effective in practice even if still hard in theory.

Appendix A

Implementation of Module in TimeWiz

Implementation of module in TimeWiz was a relatively straightforward process. TimeSys provided a set of instructions to create a <name>Analysis.dll file; for this case, the file was named UtilityAnalysis.dll. By following these instructions, a workspace in Microsoft Visual C++ was created which created the .dll file when compiled. All code written for this project was incorporated into the files UtilityAnalysis.cpp and UtilityAnalysis.h.

To implement the utility module in TimeWiz from the compiled .dll file, a few steps were necessary. First, a catalog was made through the TimeWiz Catalog Designer (named Utility.TWZ) that contained all of the desired variables as described in Chapter 6. This catalog was then moved to the Catalogs subdirectory contained in the TimeWiz directory. Finally, the UtilityAnalysis.dll file was moved to the Plugins subdirectory in the TimeWiz directory. Once these steps were complete, the utility module could be run within TimeWiz by selecting File→New→Utility.

Bibliography

[1] Open group 98 mk7.3a release notes. Technical report, The Open Group Research Institute, Cambridge, MA, http://www.real-time.org/papers/RelNotes7.Book.pdf, 1998.

[2] Omg 99a real-time corba 1.0 specification, orbos/99-02-12, and errata, orbos/99-03-29. *Object Management Group*, 1999.

[3] Real-time corba. *http://www.omg.org/cgi-bin/doc?orbos/99-02-12*, 1999.

[4] Kenneth R. Baker. *Introduction to Sequencing and Scheduling.* John Wiley & Sons, New York, 1974.

[5] Ecker Klaus H. Schmidt Gunter Blazewicz, Jacek and Jan Weglarz. *Scheduling in Computer and Manufacturing Systems.* Springer-Verlag, Berlin, 1994.

[6] Donald E. Brown and William T. Scherer. *Intelligent Scheduling Systems.* Kluwer Academic Publishers, Boston, 1995.

[7] Jensen E.Douglas Kanevsky Arkady Maurer John Wallace Paul Wheeler Thomas Zhang Yun Wells Douglas Lawrence Tom Hurley Pat Clark, Raymond. An adaptive, distributed airborne tracking system. In *Proceedings of the Workshop on Parallel and Distributed Real-Time Systems, IEEE*, http://www.real-time.org/papers/wpdrts98.pdf, 1998.

[8] Raymond K. Clark. *Scheduling Real-Time Activities.* PhD thesis, Carnegie Mellon University, 1990.

[9] Maxwell William L. Conway, Richard W. and Louis W. Miller. *Theory of Scheduling.* Addison-Wesley, Reading, MA, 1967.

[10] Lenstra J.K. Dempster, M.A.H and A.H.G. Rinnooy Kan. *Deterministic and Stochastic Scheduling.* D.Reidel Publishing Co, Boston, 1982.

[11] Simon French. *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop.* John Wiley & Sons, New York, 1982.

[12] J.R. Jackson. *Scheduling a Production Line to Minimize Maximum Tardiness.* Research Report 43, Management Science Research Report Project, UCLA, Los Angeles, 1955.

[13] E. Douglas Jensen. Real-time lexicon. *http://www.real-time.org/no_frames/rt-lexicon.htm.*

[14] E.D. Jensen and J.D. Northcutt. Alpha: An open operating system for mission-critical real-time distributed systems-an overview. In *Proceedings of the 1989 Workshop on Operating Systems for Mission-Critical Computing, ACM Press,* 1990.

[15] E.Douglas Jensen. *A Proposed Initial Approach to Distributed Real-Time Java.* The MITRE Corporation, http://www.real-time.org/papers/isorc2000.pdf, 2000.

[16] Ralya Thomas Pollak Bill Obenza Ray Klein, Mark H. and Michael Gonzalez Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems.* Carnegie Mellon University/Software Engineering Institute, Kluwers Academic Publishing, 1993.

[17] Rainer Kolisch. *Project Scheduling Under Resource Constraints.* Physica-Verlag, Heidelberg, 1995.

[18] Yoshida K. Mercer C. Rajkumar R. Lee, C. Predictable communication protocol processing in real-time mach. In *Proceedings of IEEE Real-Time Technology Applications Symposium*, 1996.

[19] C. Liu and J. Leyland. Scheduling algorithms for multiprogramming in a hard real-time environment. 30, 1973.

[20] Errol Lynn Lloyd. *Scheduling Task Systems With Resources.* PhD thesis, Massachusetts Institute of Technology, 1980.

[21] Dennis Lock. *Industrial Scheduling Techniques.* Gower Press, London, 1971.

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

[23] S.E.Shipman R.K. Clark J.D. Northcutt R.B. Kegley B.A. Zimmerman Maynard et al. 88 Maynard, D.P. and P.J. Keleher. An example real-time command, control, and battle management application for alpha. Tr-88121,archons project, Carnegie Mellon University, CMU Computer Science Dept., Pittsburgh, PA, 1988.

[24] A.H.G. Rinnooy Kan. *Machine Scheduling Problems: Classification, Complexity and Computations.* Martinus Nijhoff, The Hague, 1976.

[25] TimeSys. *TimeSys Corporation homepage.* TimeSys Corporation, Pittsburgh, PA. http://www.timesys.com, 2000.

[26] Tripacific. Tripacific software, inc homepage. 2000.