The Alpha Operating System:

Scheduler Evaluation Experiments

Jeffrey Trull
J. Duane Northcutt
David P. Maynard
Raymond K. Clark

TR #90021

Archons Project Technical Report

February 6, 1990

The Alpha Operating System:

Scheduler Evaluation Experiments

Jeffrey Trull
J. Duane Northcutt
David P. Maynard
Raymond K. Clark

February 6, 1990

Archons Project Technical Report #90021

This research was supported in part by the USAF Rome Air Development Center under contract number F33602-88-D-0027. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of RADC or the U.S. Government.

Table of Contents

Abstract

This document describes our experience with a set of experiments designed to evaluate the behavior and performance of various scheduling policies for the Alpha real-time distributed operating system [Northcutt 87]. Alpha is an adaptable decentralized operating system being developed as a part of the Archons project's on-going research into real-time distributed systems. Timely completion of an application's computational activities is one of the most important functions of a real-time system. Therefore, the proper scheduling of those activities is of critical importance.

The Alpha programming model provides simple mechanisms for an application to specify its timeliness requirements. The scheduler may consider this time constraint information when scheduling activities for execution. To gauge the effectiveness of Alpha as an operating system for real-time applications, an understanding of its scheduling facility is necessary. To this end, a study of the effects of various different scheduling policies was made. To evaluate these policies, a real-time application was created to serve as an experimental workload. The application was designed to simplify the modeling of tasks with a wide range of timeliness requirements and to provide a visual indication of scheduler performance.

Alpha provides a clean, well-defined interface between the scheduling subsystem and the kernel proper. A general scheduler framework, independent of any specific policy, has been implemented to simplify the development and substitution of different policy modules. This framework greatly simplifies the experimental comparison of different policies. To date, eight different scheduling policies have been implemented for Alpha. Five of these policies are examined in this document—preemptive round-robin, static priority, dynamic deadline, shortest processing time, and Best-Effort [Locke 86].

The experiments were designed to compare the application-level behavior of various scheduling policies under a variety of loading conditions. Each policy was judged based on criteria such as how well the application timeliness requirements were met and what fraction of the potential value of the application was obtained. The experimental results were then studied to develop empirical observations about the relative behavior of each scheduling policy and to derive an understanding of the scheduling decisions that were made.

1. Introduction

This document describes the experience of the Archons research project in evaluating a set of scheduling policies for use in the Alpha distributed real-time operating system. Alpha is unique among operating systems in the problem area it seeks to address: distributed, supervisory-level, real-time command and control. This problem area dictates some special requirements that the system must fulfill. These requirements are associated chiefly with the need to manage resources in a timely and decentralized fashion. The Alpha programming model permits the convenient expression of the timeliness requirements of an application. The manner in which the active scheduling policy uses this information determines how well the timeliness requirements are satisfied. This set of experiments examines the behavior and performance of five different scheduling policies that have been implemented for Alpha. Before examining these policies or the experiments in detail, however, it is useful to understand the nature and requirements of Alpha.

1.1 The Alpha Operating System

The Alpha operating system is unique because of the requirements imposed by its application domain. As a distributed real-time command and control system, Alpha is intended to support applications where most of the activities in the system have stringent time constraints that are a matter of correctness rather than convenience—the safety of human life and property may be dependent on the correct functioning of the system. In addition, the distributed nature of the system also places unusual demands on Alpha. Even though the system consists of a collection of physically dispersed processing elements, system computation resources must be managed so as to cooperatively perform a single mission, rather than treated (as is currently common) as a network of communicating, but otherwise independent and unrelated individual processing elements.

The Alpha programming model directly supports these needs [Northcutt 88b]. From the viewpoint of the programmer, the physically dispersed system may be logically viewed as a centralized one. The basic abstractions in Alpha include *objects*, *threads*, and *operation invocations*. In Alpha, an object is defined as a logically related collection of data and the code used to manipulate that data. The external interface to an object consists of the set of operations that may be performed on the object, and represents the only means of accessing the object's encapsulated data. The basic unit of activity in the Alpha system is known as a *thread*, which is a representation of a logical point of program execution. Threads are the entities which animate the otherwise passive objects in the system. Threads move between objects by means of operation invocations. Operation invocations are similar in some ways to procedure calls, accepting and returning parameters in much the same way.

In Alpha, there may be any number of threads executing within a single object. Furthermore, because objects may be on any physical processing element, threads may, as a result, move between the processing nodes in the system. When a thread is created, it begins its execution with the invocation of some operation on an object. The thread continues execution until this initial operation is complete, at which point it is deleted. In the process of executing any operation, a thread may invoke operations on other objects, thereby transferring the thread into a new object (which may be on a different node). If the target of an invocation is on a different node, the thread state (including the current time constraints) is transferred to the destination node.

It is important to recognize the difference between this programming model and that of systems which support the *process-message* model. In the process model, a unit of computational activity is tightly bound to a piece of code and a physical processing element. When an application is organized into objects which may be used by many system activities, it does not make sense to confine points of control to specific code segments which belong exclusively to each thread. The thread abstraction in Alpha represents the essence of an abstract computational activity, and is not burdened by unnecessary artifacts. A thread represents a locus of program control and the current attributes the activity (e.g., its time constraints). Threads move between objects without regard to their physical location, and do not incur scheduling overhead as a result of each inter-object transition.

With respect to the real-time objectives of the system, the single most important resource that the Alpha system manages is the processing cycles available to the application. Accordingly, processor scheduling has been a topic of great interest to the Archons project for many years. The concept of *time-value functions* has been developed to describe the time constraints of activities in real-time systems. Time-value functions were originally developed by E. Douglas Jensen in the context of ballistic missile defense [Jensen 75]. The concept was subsequently developed by the Archons Project students and staff at CMU, and was incorporated in the Alpha operating system [Northcutt 88a].

As a result of the previous work in this area, it has become clear that the algorithms needed to perform time-driven resource management are computationally demanding. In addition, a distributed real-time system must also be capable of dealing with the exceptions and unexpected events that may occur as a result of the application environment. Because of its great importance and demanding requirements, the Alpha scheduling subsystem has become the center of great attention and much effort has been directed towards it in the course of developing practical solutions to the problem of time-driven resource management. In Alpha, the scheduling subsystem has been carefully partitioned from the rest of the system and a separate processing element is provided for it. Furthermore, the scheduling subsystem was designed to allow the simple and straightforward substitution of different scheduling algorithms, for the purpose of evaluation and comparison.

1.1.1 Real-Time Scheduling Requirements

In real-time systems, the timeliness of a computation is as much a part of correctness as is calculating the correct value. Threats to human life and property are the expected results of failure to meet either criteria. Most current approaches to meeting application time constraints involve the use of a sufficiently large amount of excess resources to ensure that the timeliness needs are met. This viewpoint is evident in the common belief that a real-time operating system is one that has fast context switching times and rapid interrupt handling. Under this definition, any operating system running on a fast enough processor would be real-time. What is really needed, however, are scheduling techniques that make the best possible use of processor cycles, both when there are enough resources to fully satisfy the activities in the system, and when there are not enough cycles to meet all of an application time constraints.

Some current approaches to resource management, spawned from the excess resources school, involve the use of “guarantees” about resource availability. This brand of management can be the source of a great many problems, and is in itself antithetical to providing the best use of the resources in a system. It is impossible to make any guarantees about resource availability unless the system designer is willing to permit extremely urgent exception conditions to be ignored in favor of the less important ones which have been guaranteed the resources needed. Adding more resources to the system merely postpones the inevitable moment when even they will not be sufficient.

Responsiveness to time constraints should be based not on guarantees but on system software that does the best possible job (according to some meaningful application-level definition) at any given moment. This includes not only times when there are sufficient resources, but also times when unexpected events occur and there are not enough resources to handle every need of the application. Because many real-time systems are based on the belief that is possible to make guarantees about resource availability, they do not even address behavior in overload. As a result, when overloads inevitably occur, they are handled poorly. One form of good performance in overload is if the system’s performance degrades proportionally to the amount of overload. Most existing systems experience complete failure when overloads occur. If the system is designed to have so much computation power that this will almost never happen, it could represent a serious waste of resources in the normal case. If, on the other hand, the system is designed for lower performance and does not have sufficient computational resources, system failure will occur. The optimum solution seems to be that the system should have sufficient computational resources to perform the absolutely vital system functions in the worst case. This way, the system does not fail in overload, but instead postpones or does not perform the activities of lesser importance in favor of those that are vital to successful mission completion.

1.1.2 Distribution Requirements

To be most effective, a mission-oriented distributed system must be strongly connected, and must perform its activities as a whole in a cooperative (as opposed to competitive) fashion. Alpha is the first distributed operating system to be truly *decentralized*. There is no central entity whose failure would doom the system. The Alpha system provides support for cooperation on a peer level which is implemented so that it can be logically viewed by the programmer as having all the behavioral characteristics (in terms of synchronization and coherency) that a centralized system provides.

The decentralized nature of Alpha has several important consequences with respect to scheduling the application processors. The primary example involves time-constrained computations that span multiple nodes. Proper handling of these distributed computations requires that the scheduling subsystem must be made aware of the time constraints associated with a thread when it arrives at a node. If, when a thread spans a series of nodes, a failure occurs in one of the nodes in the chain, the computation performed by that node is lost and the computations based on its results are invalidated. Therefore, the head of the thread must move back to the point prior to the first failure and Alpha must intelligently schedule cleanup activities for the orphaned threads. This includes removing time constraints that were acquired on or after the broken node.

1.2 Testing Alpha Scheduling Policies

The importance of the scheduling function in the Alpha system demands that the policies used to allocate processing resources meet (as well as possible) the requirements of the application. It is of course impossible to know in advance what characteristics any given application may require of a scheduling policy. Certain characteristics have, in the project's experience, proven to be especially helpful in real-time command and control applications. Using these characteristics as a starting point, requirements for an application to run on the Alpha system to exercise each scheduling policy on each of these points were devised.

1.2.1 Test Objectives

The scheduling policies examined in this report vary widely in such characteristics as the amount and kind of information required, the computation time used in determining a schedule, and the criteria used to rank competing threads into a schedule. A number of aspects of each scheduling policy's behavior are of interest in this effort. The primary intent of this work was to evaluate the relative abilities of various scheduling policies when given a certain amount of information by the application. Also of interest is the examination and characterization of the behavior of each scheduling policy within the context of Alpha.

The policies tested here were evaluated under both underload and overload conditions. Because the policies each use a subset of the available time constraint information and because each policy makes a different use of this information, the behavior of an application can vary greatly from policy to policy and between underload and overload. It was of particular interest to discover how well the overload performance of the scheduling policies provided for graceful degradation of system function. In general, if a system is not overloaded, a scheduler should be able to create a schedule that permits all the activities in the system to complete before their critical times (the time at which completion of the activity would no longer result in any value to the system). On the other hand, there are scheduling policies that do not make use of all of the application-specified information that is available to the scheduling subsystem in Alpha, and cannot attain even this level of performance. Other policies make use of the full information available in the time-value functions, but use different criteria to determine schedules. The characterization of the resultant application-level behavior was of great interest in these experiments.

Also examined was the response of various scheduling policies to particular scenarios representing specific resource management problems. Furthermore, the sensitivity of scheduling policies to variations in their input parameters was also of interest, and experiments were carried out to determine the extent to which the values of various parameters affect the schedules generated by each policy. These experiments provide clues about how the scheduling policies respond to small variations in constraints and how each policy makes trade-offs when faced with difficult resource management decisions.

It was also intended that the results of this effort would include additional information about the effectiveness of Alpha's design and performance. One item of particular interest is the effectiveness of the separation of policy from mechanism within the scheduling subsystem. Alpha was designed with a scheduling framework into which a wide range of user-specified scheduling policies can be inserted. This process of providing a collection of mechanisms and not mandating a specific policy is a characteristic of Alpha operating system as a whole.

An important objective of this work is the evaluation of scheduling policies to find ones which exhibit the desired behavior with respect to handling time constraints imposed by aperiodic events. It was hoped that the empirical examination of the behavior of different scheduling policies would suggest what which class of policies performs the best under conditions representative of real-time command and control applications.

1.2.2 Test Application Requirements

In order to meet these objectives, it was necessary to come up with a sample application which both contained the elements of a supervisory real-time control application and could illustrate in clear terms the efficacy of each scheduling policy with regards to its ability to successfully meet the given time constraints. The application must be one where it is possible to separate the effects of the scheduling policy from the other functions which affect the behavior of the application. To meet these two objectives, the application must contain certain elements. First of all, the sample application must contain hard time constraints—i.e. there must exist activities which demand response within a specific amount of time. If the application responds to a time-critical event after that time, the response will be of no benefit to the system. In most real-time systems, there are such activities which simply *must* be completed before a certain time has elapsed. However, there also exist applications where the penalty for not meeting a time constraint may not be the loss of the system, and in fact may not be very severe at all. Such activities are known as *soft time constraints.*

Another important feature required of the test application was the presence of multiple schedulable entities. It is necessary for the scheduler to have a reasonably large number of tasks contending for processor cycles. Having multiple concurrent activities helps ensure that the scheduler must constantly make scheduling decisions, and that the effects of those decisions will be visible at the application level. A small number of activities might reduce the demands on the system to a point where there was no longer any need for the scheduler to be intelligent about allocating processor cycles. Having multiple activities that each perform a similar function but with different time constraints would make it easier to see the impact of the time constraints clearly.

2. Scheduler Evaluation System

After considerable thought, an application was devised which meets most of the criteria expressed in the previous chapter. The abstract problem chosen for these experiments is the task of keeping several bouncing balls in the air by means of moving motorized paddles. Each ball has a paddle assigned to catch it. The application program that executes on Alpha is responsible for controlling the motion of the paddles to ensure that balls are not dropped. The application program receives sensor data giving the position and velocity of the balls, and produces a series of actuator commands to move the paddles into position.

The experimental environment is composed of two major components—the *mechanical subsystem* that includes the physical environment in which the control activity exists (i.e., the balls, paddles, walls, ceiling, floor, gravity, etc.), and the *control subsystem* that includes the application program that performs the control functions, the Alpha operating system, and the Alpha testbed on which it all runs. Figure 1 illustrates the major components of the experimental environment.

In the work described here, the mechanical subsystem consists of an accurate simulation of the mechanics of the bouncing balls and movable paddles. The system's sensors and actuators are also simulated. In addition to greatly simplifying the system, simulation of the mechanical subsystem permits the rapid and accurate gathering of data, and permits the straightforward manipulation of interesting experimental variables.

There are a number of software components which comprise the experimental environment. First is the Alpha application program which meets the given problem definition—i.e., the program must attempt to keep balls airborne by moving their associated paddles (the Alpha scheduling policy under test is responsible for scheduling the threads involved in this activity). Secondly, there is the simulator which replaces the mechanical subsystem—i.e., the devices that the program is designed to control and the physical environment in which they exist. Finally, there is a human experimenter to manage the experiments (e.g., start and stop simulation runs, alter simulation parameters, etc.).

2.1 Application Program Structure

The activities which comprise the application program have a variety of timeliness requirements. The selection and ordering of the activities executed is controlled by the Alpha scheduler. This causes the effectiveness of the application program to depend entirely on the quality of the decisions made by the scheduler in Alpha. Because each of the paddles in the application is assigned to a single ball, there is a separate system activity (i.e., thread) assigned to control the movement of each paddle. Therefore, when a paddle moves, the observer can know that the system activity responsible for the movement of that particular paddle has been scheduled and run.

The time constraints associated with threads are distinguished from each other in three ways. The first way is through *deadline* information about the current task (in this application, the time until the assigned ball falls to the level of the paddle). The second way that time-critical activities are described is by their expected computation time. This estimate is the amount of processing resources expected to be required by a thread in order to meet a time constraint. Finally, there is an

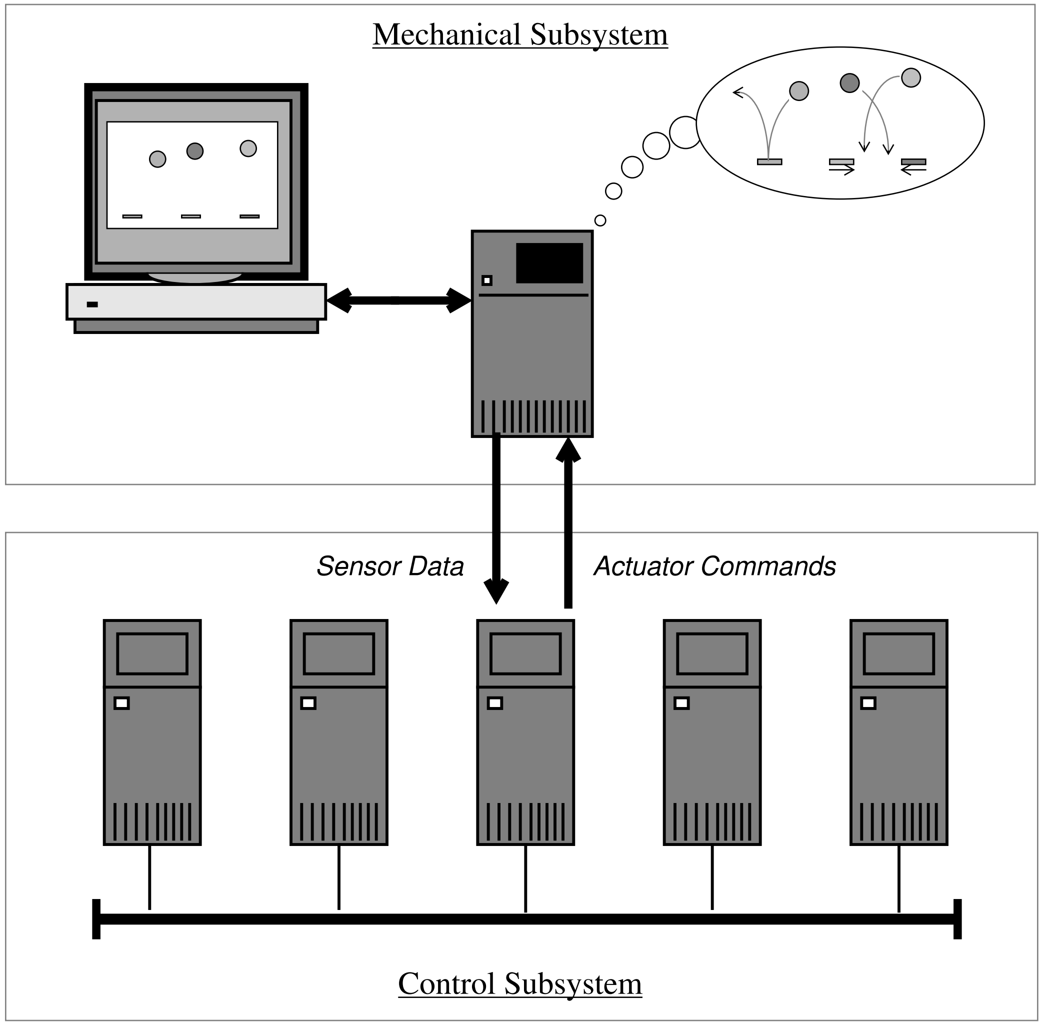

Figure 1: Experimental Environment

integer value that can be assigned by the experimenter to each thread. This value becomes a characteristic known as the *importance* of the thread. In these experiments it corresponds to the importance of the ball that the thread is attempting to keep aloft. These three characteristics of each paddle movement activity (i.e., thread time constraint) are provided by the application program to the Alpha scheduler. This time constraint information is provided so that thread execution schedules can be created based on the time a ball will take to fall to the floor, the time it will take to move its paddle to the ball intercept point, and the value of the ball to be caught. All three of these characteristics may be observed and controlled by the experimenter, so both the basis of the scheduler's decisions and the effects of the decisions themselves (i.e., the sequence of paddle movements) are directly visible.

The simplicity of the application limits the amount of application-level interference that could mask the effects of scheduling decisions. A single piece of application code can describe the correct operation every paddle activity in the system. It is only necessary to describe how, for a single paddle, to use the sensor information on the location of the ball to compute and execute a sequence of paddle movements that will place it in position to intercept the falling ball. The only difference between paddle control activities is the critical time, expected computation time, and importance information associated with each thread. The decisions about which balls to (attempt to) catch, and in what order, are as a result made entirely by the Alpha scheduler.

The simulator's interface to the application is provided through the Sun-UNIX remote procedure call mechanism. The simulator communicates with the application paddle management system to ensure that the tasks which move the paddles are running when the simulation is started, and to remove them when the simulation is stopped or the user removes the corresponding ball. To remove or add a paddle, the operator interface simply calls the corresponding remote procedure. These remote procedure calls are translated into Alpha invocations by the Alpha external communications interface. Requests for simulator operations begin as invocations from the application and arrive at the simulator as remote procedure calls. The translations between each of these systems is provided by interfacing software. Thus an invocation of a simulator interface operation eventually is performed as a remote procedure call on the Sun-UNIX system, and a remote procedure call of a paddle management routine becomes an invocation on an Alpha testbed node.

There are two major operations which tasks in the application can request from the simulator. These operations are GetSensor, which returns information concerning the current position and velocity of a given ball, and MovePaddle, which allows the application to move a paddle one increment to the left or right. MovePaddle requires a fixed amount of node computation time to run, as would be expected from an operation that required the continuous pulsing of a stepper motor. GetSensor requires some time for the data to arrive, so it involves blocking and waiting for the packet to arrive. Both operations were designed to model a physical system. There are also two major operations that the simulator can request the application to perform. These operations are AddPaddle and RemovePaddle. The application has a *paddle manager* which implements the paddle control operations that the simulator can invoke. The paddle manager allows the evaluation environment package to remotely create and delete threads that implement individual paddles.

The implementation of the paddle manager depends on the task management facilities available on the application system, as does the implementation of any auxiliary modules used to communicate with the evaluation environment's system. The design of the paddle tasks, however, is fairly straightforward and is mostly independent of the system on which they run. Each paddle task must query the position and velocity of its target ball, must calculate the predicted landing location, and must move the paddle to that location. Once the ball bounces or is dropped, the process is repeated. An important fact to realize is that the cyclic nature of this task does not make the application a *periodic* process in the sense of having predictable time constraints. The timeliness and processing requirements of the paddle tasks may be different each time they repeat the sense-move loop. As in most supervisory real-time problems, the workload cannot be predicted in advance.

Two implementations of the application were developed. These were on a Sun UNIX system and the Alpha Release 1 testbed. The UNIX version was implemented to simplify testing the application interface to the simulator. The Alpha system was the object of the actual experiments.

2.1.1 UNIX Implementation

The structure of the UNIX implementation is a direct translation of the English description given above. The paddle tasks, which are implemented with UNIX processes, use the remote procedure call mechanism to access the simulator GetSensor and MovePaddle operations. Each paddle process calls GetSensor to obtain the position and velocity of the ball as well as the distance it can move in each paddle movement increment and the time each move takes. It then computes the distance that it must move to catch the ball at the intercept point, converts to the number of movement increments that it will require, and calls MovePaddle that many times, waiting the appropriate delay between each move. If the paddle arrives early, it uses the UNIX interval timer facility to block until the intercept occurs.

There were several unfortunate characteristics of this UNIX-based implementation. Probably the major problem was that UNIX, not being a real-time system, had no facilities for describing the time constraints of a task. Each paddle task had a time constraint which was the time required for a ball to fall to the level of the paddle, after which there was no use in continuing to move to the intercept point. There was no way to describe this time constraint to the system, and there was no way to tell the system that such an movement attempt should be aborted after it had already failed. It was possible to use the interval timer facility and a variety of complicated tests to generate the desired behavior. However, there was no general, natural way of describing the characteristics of the paddle tasks and having them considered as part of the scheduling process.

2.1.2 Alpha Implementation

Because there are general mechanisms for describing time constraints in the Alpha system, it was straightforward to implement the paddle tasks in a way that permitted them to describe their time constraints to the scheduling subsystem. The structure of the paddle task remains much the same from the UNIX version. However, the paddle task now used the information acquired from the GetSensor call to determine the time required to move the paddle to the intercept point. The importance of the task for the purposes of the scheduler is given by the value of the ball scaled by a constant. The value function is given by a constant which drops to zero at the time that the ball will pass the level of the paddle. Such a time constraint forms what is called a *hard deadline*.

Depending on the scheduling policy used, some or all of this time constraint information may be utilized to determine a good schedule. Under policies that support dynamic deadlines, the tasks can be automatically aborted out of their deadlines once the deadline has passed (this eliminated the elaborate checking to determine if the ball has passed yet). As soon as the ball is dropped, the deadline is aborted.

2.2 External Environment Simulator

The external environment simulator serves as a surrogate for the real-world sensors and actuators comprising the external environment of a supervisory control system. It provides functions to read the sensors and move the actuators, and keeps track of the changes in the simulated environment. A graphical operator interface allows the user to control the simulation. These two modules together allow the experimenter to manipulate the situation the workload and record the results.

2.2.1 Simulator Structure

The simulator models the motion of the balls and the paddles and provides sensor information about the current state of both. The interface between the evaluation environment and the application was designed so that the application could be placed into a real, physical situation identical to that which was being simulated, and no change to the application would be necessary. In addition to modeling the motion of the balls, the simulator accepts requests from the application that cause the paddles to move. It also accepts commands from the operator interface that allow the user to start and stop the simulation, create or modify balls, and extract performance information as the application is running. Other actions performed by the simulator include notifying the application when a ball has been dropped.

The simulator performs most of its work during the simulation update period. This occurs once every time period, and involves updating the positions and velocities of all the presently existing balls, as well as checking for collisions and performing bounce calculations. These computations are performed by modeling elastic collisions with the walls and paddles. To better simulate the kinds of aperiodic loads which actual supervisory real-time systems experience, the bounce direction from a collision with the paddle is selected randomly. If during the process of updating the simulator state, it is discovered that any balls have been dropped, the application is notified and directed to remove the associated paddle. However, to enable the user to sustain a constant load on the system, it is possible, using the operator interface, to select certain balls that will remain in the simulation when dropped and will bounce back up above the paddle instead of being removed from the simulation. The output of the simulator includes timestamped messages indicating when balls are caught or dropped, and what the value of each ball is. With this information it is possible to compute a wide variety of useful statistics about the performance of the system.

2.2.2 Operator Interface

The operator interface provides the experimenter with control over the simulation. It displays a window that contains a small control panel for managing the simulation and a larger subwindow that shows the current simulation state (see Figure 2). There are buttons on the control panel that permit the user to stop and start the simulation. When in the stopped state, no paddle tasks are running on the Alpha system and the user may add or remove balls and paddles without difficulty. Balls are added or removed by using two buttons reserved for that purpose. The operator selects locations by “clicking” the mouse button in the display area to indicate the ball or the position desired. Adding or removing a ball will result in the addition or removal of a corresponding paddle. In the stopped state, however, no paddles are shown; the paddles first appear when the simulation is started. There are also facilities for capturing and replaying sequences of action that appear in

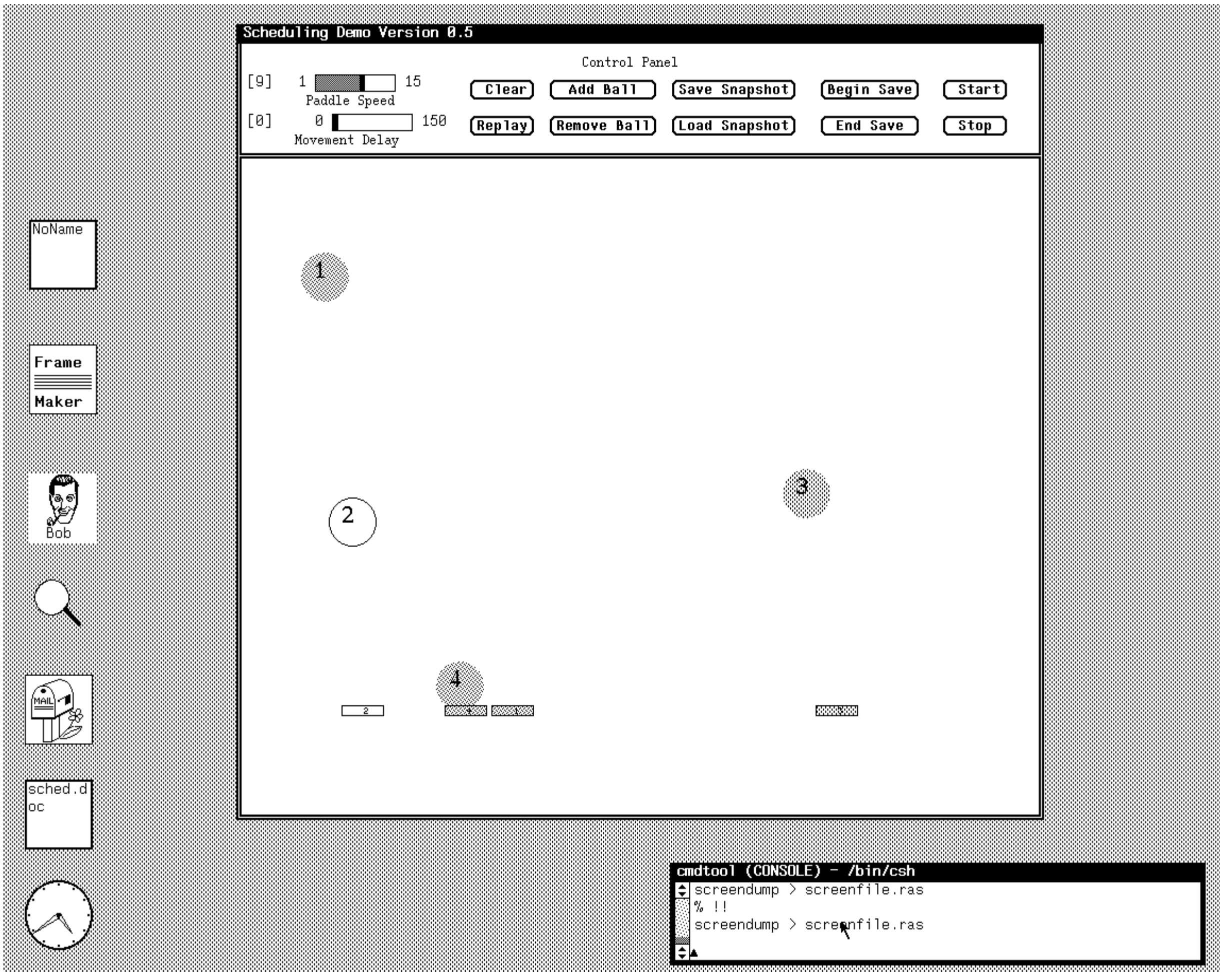

Figure 2: User Interface

the display window, as well as for saving and restoring initial ball placements. Selecting a ball displays a panel that allows the user to alter its velocity, height, value, and paddle position.

When the user "clicks" the start button, the simulation begins and the balls start moving according to their given initial conditions. Once the simulation is started the paddles start moving to catch the balls. The user may alter the distance a paddle travels with a single pulse of the stepper motor, the *paddle speed*, and how much compute time each pulse takes, the *movement delay*. With these two controls, it is possible to change the amount of load present on the system. For example, increasing paddle speed decreases the number of computations required. Increasing the motion delay increases the computation time required. This ability to adjust the positions and velocities of the balls and paddles greatly simplifies the task of experimenting with a variety of loading conditions.

3. The Structure of an Alpha Scheduler

The scheduling function in Release 1 of Alpha is managed by an independent subsystem that executes on a separate processor. A formalized interface defines the messages passed between the Alpha kernel and the scheduler. To simplify the development of scheduling policies, a general framework for implementing policies has been defined. The framework handles the actual message generation and processing, leaving the individual policy to decide how to respond to various messages.

The scheduler framework dispatches each kind of message from the application processor to a policy-supplied handler. Examples of these messages include messages that indicate that a task should be added to or removed, messages that define how the time constraints of the currently running task have changed, and others which support the distributed nature of Alpha, including clean-up for certain aborted operations. Each scheduling policy may choose how to respond to these messages and how to utilize the information provided therein. All eight schedulers which were implemented within this framework required only subsets of the information provided by these messages, and none needed any structure which was not easily created within this framework.

3.1 Scheduler/Kernel Interface

The function of a scheduler in the Alpha system is to determine which of the currently ready threads to run. To make this decision, the scheduler may use the time constraint information provided to the scheduler by the application. This information consists of three parts: a *time-value function*, indicating the time-varying value of completing a task at a given time, an *expected computation time*, the cycle time required to complete the task, and an *importance*, a value used to scale the value function. A scheduling policy must handle changes in these thread-provided parameters and, if necessary, correctly reevaluate the currently active schedule.

In Release 1 of Alpha, the scheduling subsystem executes on a separate processor from the application (and most of the Alpha kernel). The scheduling framework implements a queued, message-based communication channel between the processors. Messages are transmitted in two parts. There is a command part, which indicates the command to be performed, and a body which gives parameters to the command. The commands are:

- **Add:** used to tell the scheduler of the presence of a thread which is newly available for scheduling. This can occur when a new thread is created, when a blocked thread is unblocked, or when a thread from another node starts running on the node. This last variant is referred as a “surrogate add” because the structures representing the thread on this node are acting as surrogates for the thread structure on the node where it was created. There is also a version known as a *delayed add*; this provides a time delayed Add command and is utilized in implementing the kernel Sleep operation. The parameters describing the function include a unique identifier for the scheduler to refer to in communications with the application processor. If the Add is a surrogate add, i.e. if the thread was created on another node and its point of control has just arrived on this node, the parameters include information about the time constraints it accumulated prior to arrival on this node.
- **Remove:** indicates that the current thread has voluntarily given up the processor. This

usually occurs when the thread which is currently running has blocked awaiting some activity. Another version called Kill is used when the currently running thread has permanently given up the processor. This occurs when a thread returns from the invocation it started in, and when a thread that originated on another processor completely finishes its work on the node.

- **Change:** This command arrives when the currently running thread has altered its time constraints in some way. There are three ways in which this can occur. First, the thread may enter a new deadline. This is referred as *pushing time constraints*, because deadlines may be nested. Second, the thread may exit a deadline. For similar reasons, this is referred to as *popping time constraints*. Finally, the thread can change its importance.
- **Access Scheduling Information:** There are four commands of this type. They all relate to conditions when the time constraints must be accessed or altered outside of the normal stacking methods. The *dump* subcommand is used to gather a thread's time constraints in preparation for a remote invocation. The *update* command is used to update time constraint information such as total computation time after the thread has been executing on a remote node. There are two commands that deal with abort conditions that permit the correct time constraints for abort processing to be determined. These are the *Get* and *Set Abort Scheduling Information* commands. If a deadline aborts, the Get command is used to determine the correct time constraints to operate the abort cleanup code with. The Set command is used to force the time constraints of a specific thread into those which are required for its abort processing.
- **Preemption Confirmation**: This is a necessary element of the communication between scheduler and kernel. It is sent after a request to replace the currently running thread with another has been successfully processed. This allows the scheduler to update the amount of time a thread has run. There are two variations of this command, one of which indicates that the currently running thread was preempted successfully, and one which indicates that the processor was idle when the preemption was requested. This message is necessary to provide the scheduler with a consistent picture of what is happening on the application processor.
- **Statistics Control**: There are two of these commands, that turn on and off optional statistics gathering, if the current scheduling policy supports logging.

There are only a few circumstances in which the scheduler must initiate communications with the kernel. Several of these instances occur as part of handshaking involved with the scheduler commands. Two, however, are of fundamental importance to the operation of the system. The *Preempt* command, which has been previously discussed, is used to tell the kernel to run a specific thread. The *Abort* command is used when the scheduler determines that a thread's critical time has passed. It initiates abort processing for the missed deadline. This command is, of course, only of interest for policies that support deadlines.

3.2 Structure of Generic Scheduler

Each of the scheduler commands described in the previous section must be implemented by every scheduling policy. The policy routines which implement these commands are automatically called when the commands are received. It is possible to supply a null function if the command is

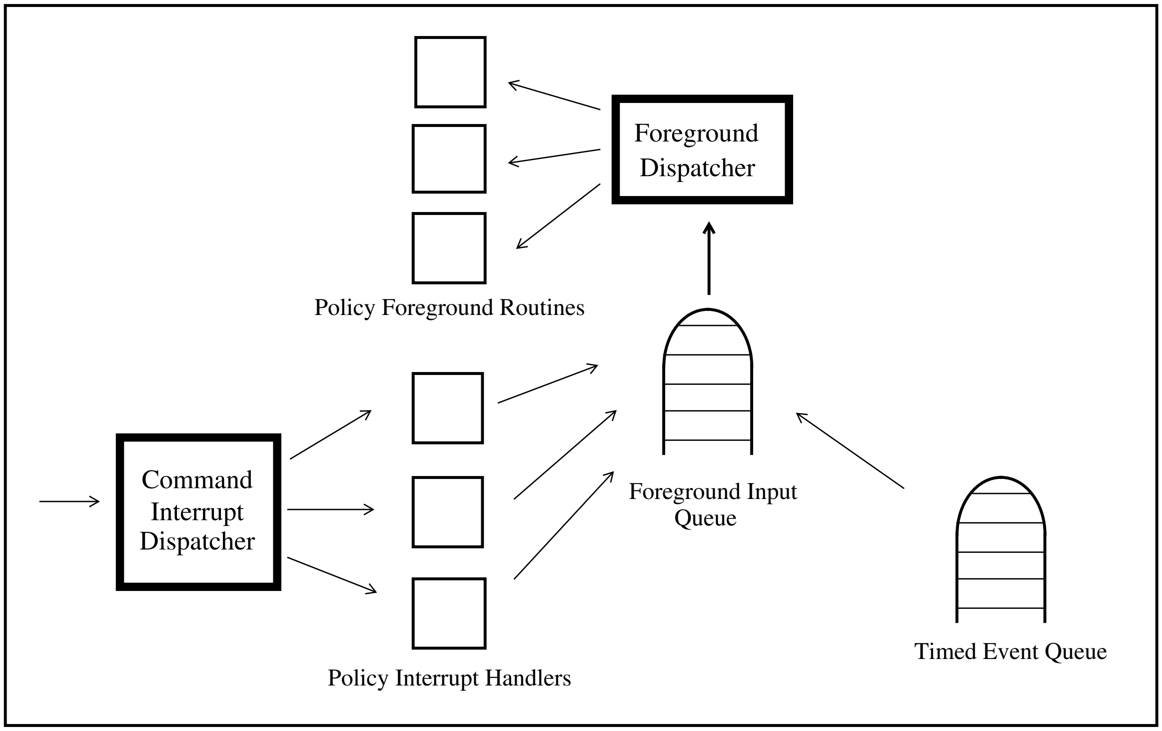

Figure 3: Basic Scheduler Structure

to be ignored by the policy, or an error function if the command should never be received by the policy.

Each policy must supply two types of command handlers. The first kind is called when the command is received under interrupt, and allows rapid handling of messages as well as general interrupt-level processing. In addition, the scheduler framework supplies an internal message queue which may be used to queue commands up for foreground processing (see Figure 3). The framework calls foreground procedures corresponding to the commands in the queue as they emerge. Any part of the handling for a command may be performed either under interrupt or in the foreground. If all the commands are to be handled in the foreground, then the interrupt tasks would simply enqueue the incoming commands into the foreground queue.

The minimum requirements for a scheduler in the Alpha system are fairly simple. The scheduler must remember the set of ready threads, and if the set is non-empty, make sure that there is always something running on the application processor. The scheduler framework provides a queue package which aids in the manipulation of schedules.

Scheduling policies that are more sophisticated, generally share some common characteristics. First, the schedulers generally construct a list describing the order in which threads should run. When a thread is added to the ready list or the time constraints of a thread are changed, some computation is required to determine whether or not to reorder the list. The operation of other commands, if any, depend on how the policy utilized the scheduling parameters provided by the threads.

4. Scheduling Policies

Five of the eight policies implemented for use in the Alpha system were selected for evaluation in these experiments. The following sections briefly describe the operation of each of these policies.

4.1 Round Robin

The Round Robin policy is the fairest of all the schedulers. It treats all threads in the system equally, giving each eligible thread the same fraction of the processor time.

The implementation of the Round Robin policy is straightforward. Every 100ms a timer event is inserted into the queue by the policy timer interrupt routine. Upon receipt of a timer event, the foreground command processor calls the timer routine, which removes the element at the head of the queue and inserts it at the end. If the new head is different from the old head, the application processor is preempted with the new head of the queue. Add and Remove commands are handled in the simplest possible way. If the command is an Add, the new thread is added at the end of the ready queue. If the command is a Remove, the thread is removed from the ready queue, and the application processor is preempted with the new head of the queue. Under Round Robin, time constraints are ignored, as are the miscellaneous commands associated with accessing scheduling information.

4.2 Static Priority

Static Priority scheduling is the simplest method of dealing with scheduling parameter differences between the threads available to run. Each thread is assigned a priority which is equal to the importance part of its time constraint. The scheduler always selects the thread with the highest priority, or one of them, if there are several with the same importance to run. The time-criticality of the thread not taken into account.

The implementation of Static Priority requires no timer usage. The scheduler framework's queue package includes ordered insertion routines, so the Add handler simply makes an insertion into the ready queue, then preempts the application processor if the head changes. The Remove handler removes the thread and preempts with the new head of the queue. The Change handler handles requests to change the importance of threads. This change is accomplished by removing the thread from the queue and reinserting it with the new importance. Again, if the head of the queue changes, the application processor is preempted with the new head of the queue.

4.3 Deadline

The Deadline policy always schedules the thread with the closest critical time. For this implementation, the Deadline policy was extended to provide for the abortion of deadlines that have expired. If a thread misses a deadline, it is forced out of the deadline section of code and begins executing a user-defined deadline abort handler. If the system has enough cycles to meet all of the application deadlines, the Deadline policy is optimal—every time constraint is satisfied. If there exist deadlines which cannot be met, which is what is meant by *overload*, then the Deadline policy may perform very poorly. For example, this policy may attempt to run threads which cannot meet their deadlines, but which have early deadlines. The deadline policy is implemented as a priority scheme where the time to deadline forms the priority and the head of the queue is the item with the lowest priority. However, the Deadline policy uses the timer facilities to make sure each thread is aborted when its deadline passes.

4.4 Shortest Processing Time

The Shortest Processing Time policy considers only the required computation time for each thread. SPT always selects the thread with the lowest remaining computation time to run. It is similar to Deadline and Static Priority in that it uses only a single figure of merit to determine a schedule. It lacks optimal behavior of Deadline in underload, but may perform well under overload since it is more likely to pick threads which can be performed. SPT in general attempts to maximize the system throughput by completing as many threads as possible per unit time. SPT as implemented for Alpha has the automatic deadline abort mechanism mentioned above. The implementation of SPT is almost identical to that of Deadline, except the schedule queue is ordered by the remaining processing time.

4.5 Best-Effort

Given the problems with the other schedulers, which use only a small portion of the available information to construct a schedule, it is clear that any superior scheduling policies will have to make decisions based on all the information in the time constraints as well as the expected computation time. It is known that Deadline scheduling is optimal when the system is underloaded. An improved scheduler can therefore use a Deadline scheduling method when the system is underloaded. If the system is overloaded, the scheduler must decide which threads *not* to run. Threads that cannot complete their deadlines are one obvious choice. If that is not sufficient, the scheduler chooses those threads whose completion would be the least useful to the system. One metric of utility to the system is called the *value density* of a thread. The Archons project has developed a scheduling policy, called the Best-Effort policy, that uses value density as a metric when shedding load. Refer to [Locke 86] for a complete description of the Best-Effort algorithm.

The implementation of the Best-Effort policy maintains a queue of threads in deadline order. As each new thread is inserted, the policy checks to see if overload has been reached. When overload is reached, the policy sheds load by removing the least valuable threads until the system is no longer overloaded.

5. Experimental Results

This chapter describes the experiments performed on the chosen scheduling policies and their results. Justification is provided for the particular experiments that were performed, and the results of the experiments are presented.

5.1 Experimental Design

The goal of these experiments was to develop an understanding of the behavior and performance of the schedulers described in the preceding chapter. The experiments performed provide a broad range of quantitative information about each of the schedulers used. By varying the loading conditions and analyzing the resulting data, many different metrics could be extracted to compare the policies.

5.1.1 Load Generation

It is possible to increase or decrease the number of threads in the application, by changing the number of balls in the scenario. Having a large number of threads tends to make the tests more accurate since small variations in the behavior of the threads have less of an effect on the large-scale behavior of the application. With more threads it is easier to ensure that the scheduling policy being tested always has ready threads to choose from, thus improving the quality of the information extracted from the test.

The loading characteristics are also affected by the initial state of the balls and paddles. It is possible, but in general not desirable, to cause the conditions at the start of the simulation to be more or less favorable by placing the paddles farther away or closer to the balls they are catching, increasing the velocity of the balls, etc. This type of testing was done to analyze individual decisions made by each policy as an aid to understanding their behavior, but is detrimental to observing long-term effects since it introduces start-up transients that obscure the steady-state performance.

There are two ways to increase or decrease the computation time required by each thread in the system. The first method is to change the paddle speed. Increasing the paddle speed reduces the number of times each thread consumes a block of processor cycles. Decreasing the paddle speed has the opposite effect. The other way to change the computation time is to alter the movement delay. Increasing this parameter will result in increased computation by each thread since more effort is required to move the paddle each step on its way to the intercept point.

5.1.2 Evaluation Metrics

There are several factors which must be considered in order to make a fair comparison between scheduling policies. The different policies have implementations which may vary considerably from that which may be optimally attained. Thus, it is important to factor out the various problems that are due only to the implementation of the policy. The primary factor which affects the apparent performance of the schedulers is the fact that some schedulers may take longer to provide a thread for the application to run after the currently running thread blocks. This effect is primarily realized in the system throughput, resulting in an apparently faster application. Part of the data collected is a measure of the system throughput, the total number of paddle movements accomplished in each minute. It is possible to utilize this throughput measure to normalize all of the schedulers to a common measure, which was selected to be the value to the system accomplished by the Round Robin scheduler. To determine the normalized value to the system obtained by a given scheduler, it is only necessary to divide by the throughput measure of the policy and multiply by the throughput measure of Round Robin. In other words, the data is interpreted as though it came from a scheduler which had the throughput of Round Robin, but made different decisions. This enables the examination of the *decisions* made by the scheduler independent of the quality of the policy implementation.

The time-value curve of a system activity is composed of information on how the completion value of a segment of code varies with time. In combination with the importance, it can provide the scheduler with knowledge of how much value the system would accrue from the computation if the activity was scheduled at a certain time. A scheduling policy should in theory be able to determine the most valuable schedule possible from this information. However, some policies use only subsets of this information, and use the information in different ways, and thus may create other schedules of varying worth. The aggregate value to the system created by each scheduling policy's allocation of computing cycles is the primary metric used to compare policies in this report. Thus the main criteria we will use as a metric to compare scheduling policies is the extent to which each policy maximizes the value provided to the system by the application. This is the logical basis on which to judge schedulers, since a scheduling policy is intended to translate as best as possible the characteristics of each thread into a good schedule for the system. Since the set of characteristics provided by threads in Alpha is the completion value as a function of time, then the ideal scheduler is one that selects the schedule that provides the greatest value. Over long periods of time, the better policies will provide more value to the system than inferior ones.

The second major metric used to compare policies is the percentage of the time constraints met by each policy. This metric is of secondary importance since maximizing the number of met time constraints does not necessarily maximize the value obtained for the system. Nevertheless, it may provide clues as to why different policies perform as they do.

5.2 Behavior Analysis

To understand why each scheduling policy behaves as it does, it is helpful to examine some specific scheduling scenarios and to analyze how each of the schedulers would respond.

The first scenario of interest is one where all the balls may be caught, but only if a certain sequence is followed. This condition is shown in Figure 4. In this picture, ball A takes longer to return to the intercept level than ball B. Static Priority will fail in this case since ball B bounces twice for every single bounce of ball A. If one were to assign a higher priority to ball A, ball A would always be caught first and ball B would be missed on its second bounce. If one were to assign a higher priority to ball B, A would be missed. Both Deadline and Best-Effort, on the other hand, would catch ball A, then ball B twice, then ball A again. SPT would give a higher priority

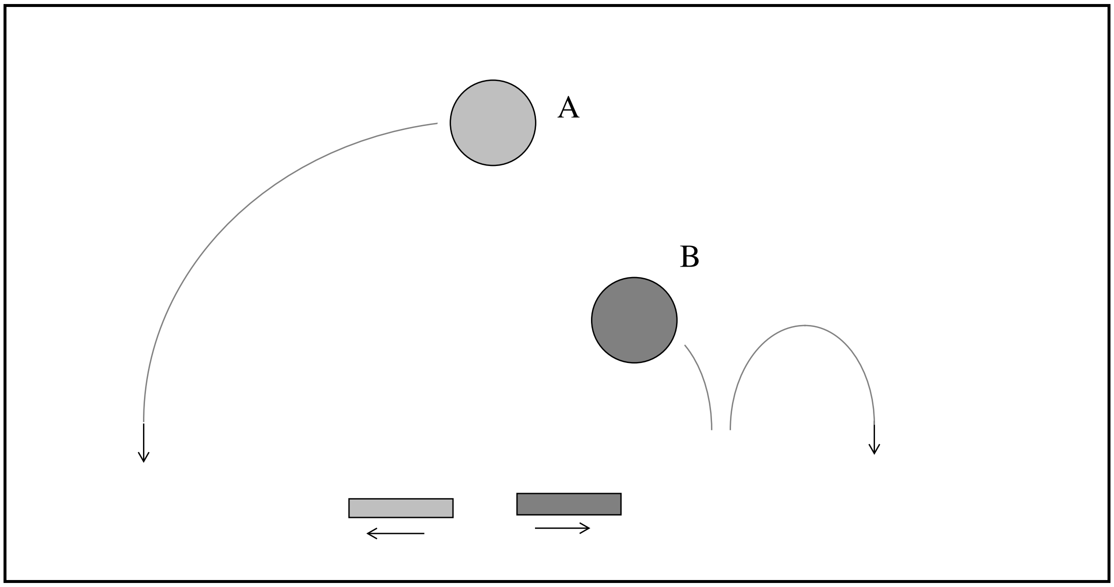

Figure 4: Dynamic Priority Example

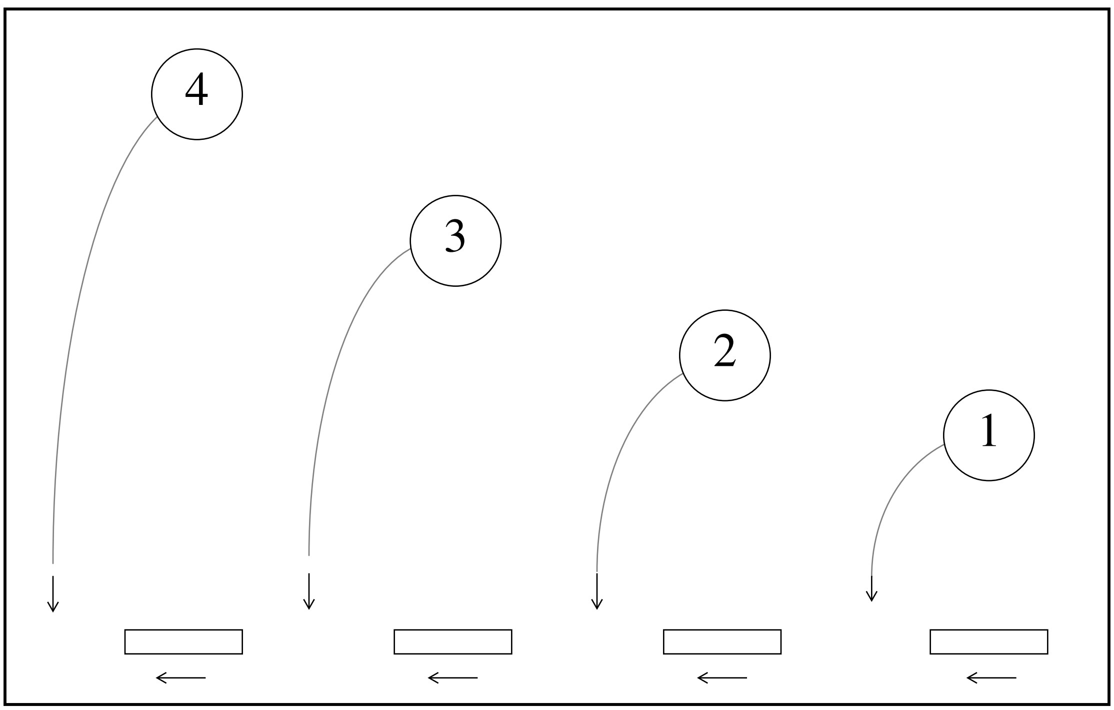

Figure 5: Inverted Deadline/Value Example

to catching B, which takes less computation time, and so would work exactly like Static Priority when the higher priority is given to ball B.

Another underload scenario is shown in Figure 5. There are four balls with values 1 through 4. The value 4 ball is farthest from its intercept point and value 1 ball is the nearest. The paddle

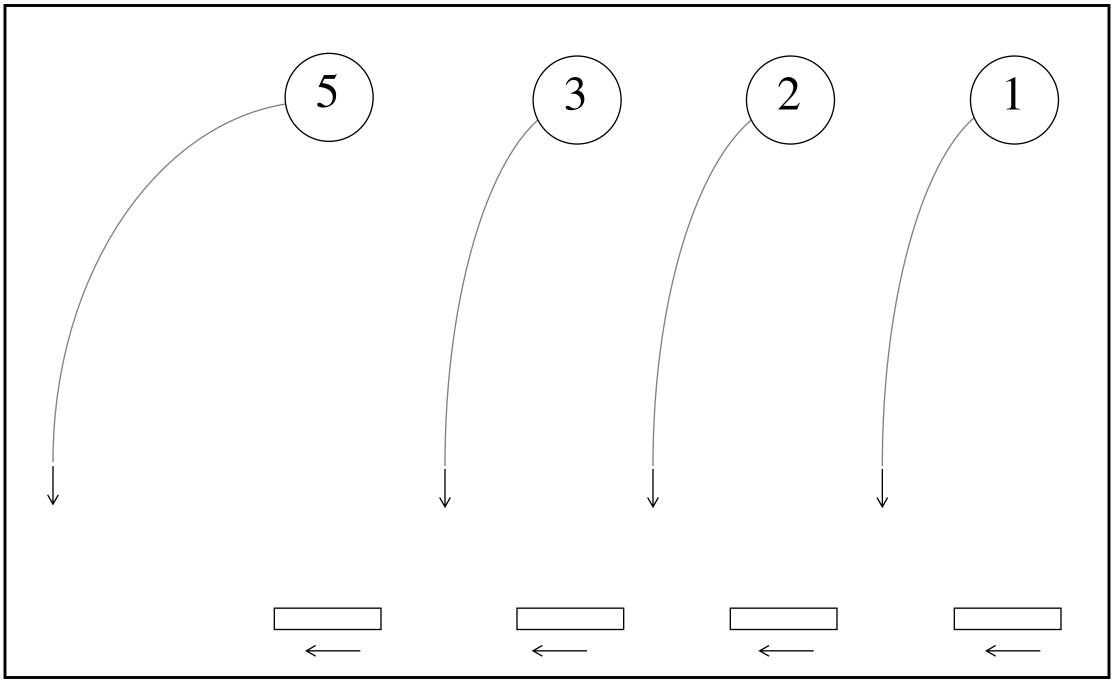

Figure 6: Simple Overload Example

assigned to each ball is one time unit away from its intercept point and each ball is as many time units away from the intercept point as its value. In this case, the urgency of each computation is the inverse of the importance of the computation (i.e., the higher the value of the ball, the less urgent it is to catch it). Static Priority will fail badly on this scenario. It will first move to intercept the value 4 ball, thus dropping the value 1 ball. It will then move to intercept the value 3 ball and drop the value 2 ball. Static Priority will successfully catch the balls of value 3 and 4, but will drop those with value 1 and 2. Both Deadline and Best-Effort will catch all of the balls in this scenario, while SPT will act in random order (all the paddles take one time unit to move into position).

The next case of interest is one where it is not possible to catch all of the balls (Figure 6). The value 5 ball is three time units away from the intercept point, as is its paddle. The value 3, 2 and 1 balls are also each three time units away from their respective intercept points; but their paddles are each one time unit away from the intercept point. Static Priority would successfully catch the value 5 ball, dropping balls of value 3, 2, and 1 in the process. Best-Effort would recognize the overload situation and abort the catching of the value 5 ball in favor of catching the value 1, 2, and 3 balls that provide more value to the system. Deadline would catch two of the three smaller value balls, then select randomly between moving one time increment toward the value 5 ball (which now cannot be caught) or catching the third small ball. SPT would, in this case, successfully catch the three small balls, because they all require fewer computational resources to complete.

Another interesting scenario is shown in Figure 7. This figure represents another overload situation. All three balls will arrive at their intercept points at the same time (two time units). Each of the paddles is only one time unit away from their intercept points. Both Best-Effort and Static

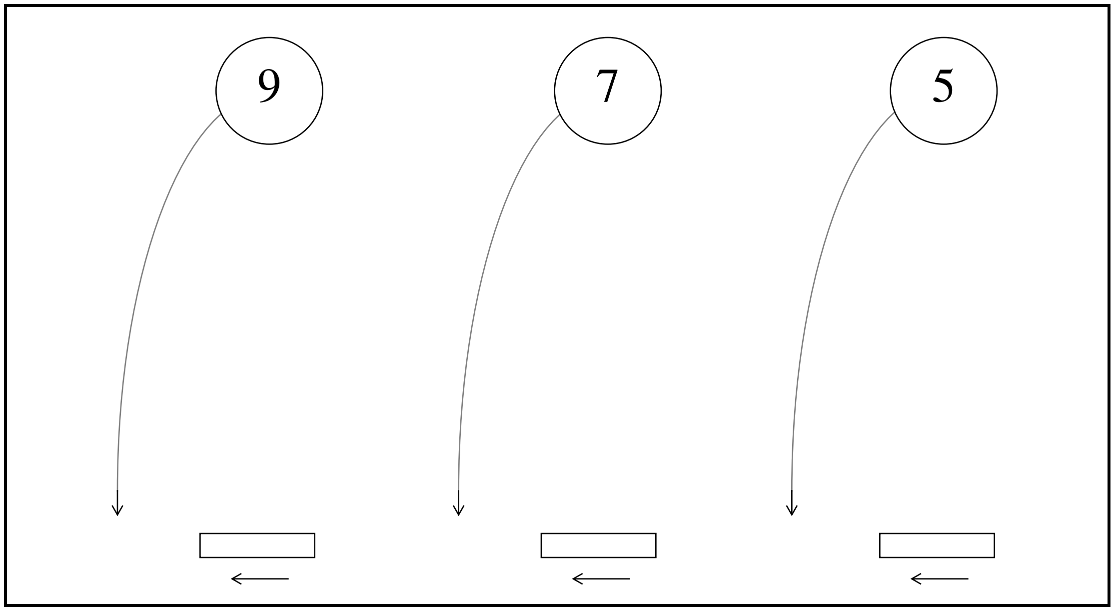

Figure 7: Value Selection in Overload

Priority will select the value 7 and 9 balls, while both SPT and Deadline will select randomly (all have the same deadline and computation time).

Finally, there is the case in which one ball is completely impossible to catch. Such a scenario is shown in Figure 8 where a ball of value 3 is one unit of time from its intercept point, while its paddle is two time units away. There is another ball of value 1 which is one unit of time away from the intercept point, as is its paddle. Static Priority will try to catch the uncatchable ball of value 3 while disregarding the catchable value 1 ball. Best-Effort will abort the paddle of the value 3 ball since its deadline cannot be made, and will catch the value 1 ball. Deadline will choose randomly between the two balls (which have the same deadlines). SPT will catch the ball of value 1 since it requires less computation time.

These scenarios indicate that the Best-Effort policy behaves will in a variety of different loading conditions. Unlike other policies which may perform well in a limited domain, the Best-Effort policy uses the full information available in the time-value function specifications to intelligently schedule tasks in both underload and overload situations.

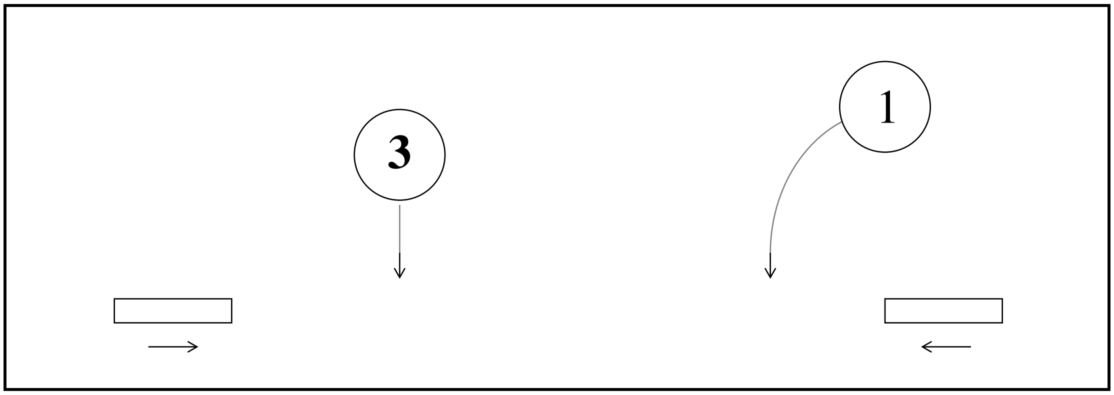

Figure 8: Impossible Time Constraint

5.3 Simulation Results

Several metrics for long-term performance were collected from each scheduler. The first measure examines how sensitive each scheduler was to variations in thread importance (i.e., ball value). The second metric records how successful each of the scheduling policies was at meeting the application time constraints (in terms of percentage of time constraints satisfied). The final indicator combines both ball value and time constraint measures to judge the overall performance of each policy.

5.3.1 Thread Importance Sensitivity

As may be seen from the graphs of percent caught versus ball value (shown on the following pages), each scheduler has a characteristic form indicating the trade-offs it makes when catching balls of different values. Round Robin, for example, has a flat graph (Figure 9) indicating that it does not distinguish between threads based on their value. Each thread in the system receives the same fraction of the available processor cycles. As expected, increasing the paddle speed increases the application's ability to catch balls. The Deadline algorithm (Figure 10) is also not responsive to ball value and shows similar behavior.

The Shortest Processing Time algorithm (Figure 11), which also disregards thread importance, shows some sharp differences between ball values. This effect is due to the particular way in which the application interacts with this scheduling policy. The amount of movement, and thus the compute time, needed to catch a given ball often increases after the ball is missed. The paddle needs to move farther to make the next catch, has a greater required computation time, will thus be less likely to be scheduled, and so also less likely to catch the ball. If, on the other hand, the paddle catches the ball, it will be closer to the intercept point for the next catch, will require less computation time, will be more likely to be scheduled, and will be more likely to again catch the ball. Thus, balls, once caught, are likely to continue being caught. Once dropped, they are likely to continue being dropped. Therefore each ball will tend to either be caught well or caught poorly for most of each run. This explains why the SPT algorithm shows such a wide variation in catching percentages with respect to ball value. Most applications do not exhibit this type of behavior.

.

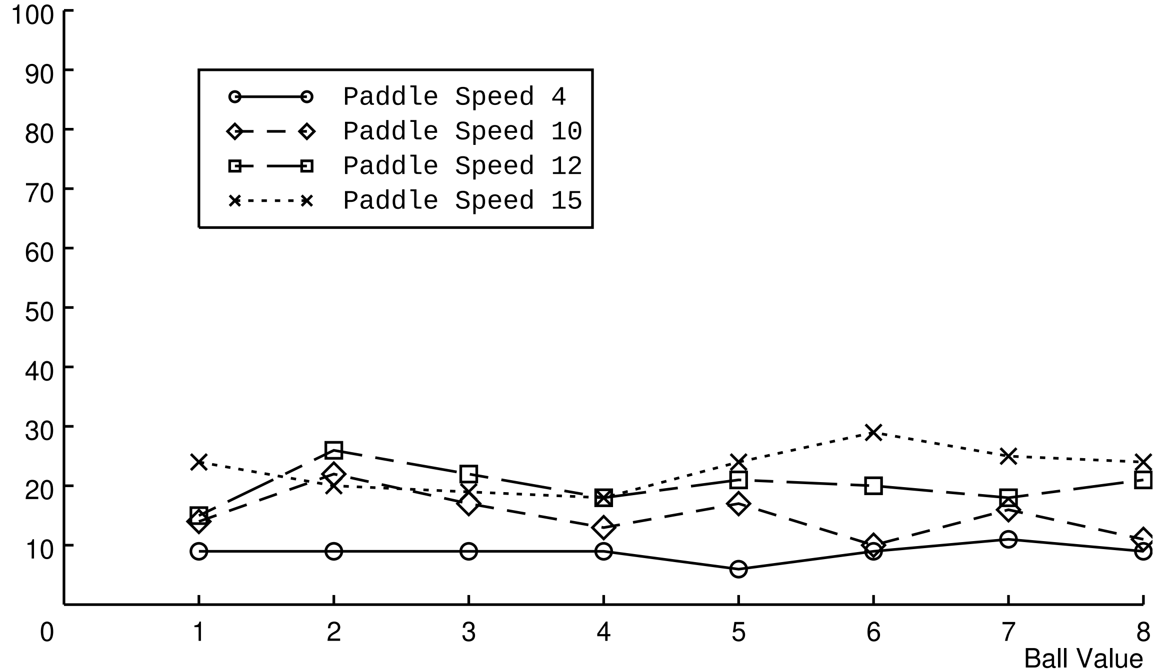

Figure 9: Round Robin Percent Caught

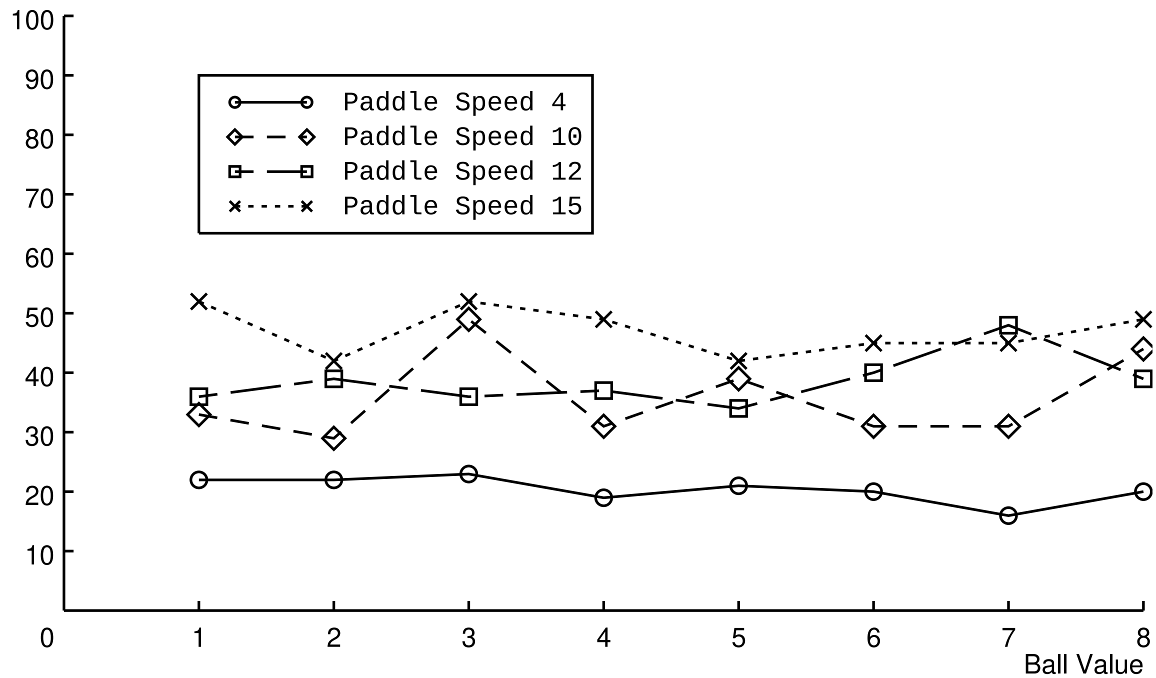

Figure 10: Deadline Percent Caught

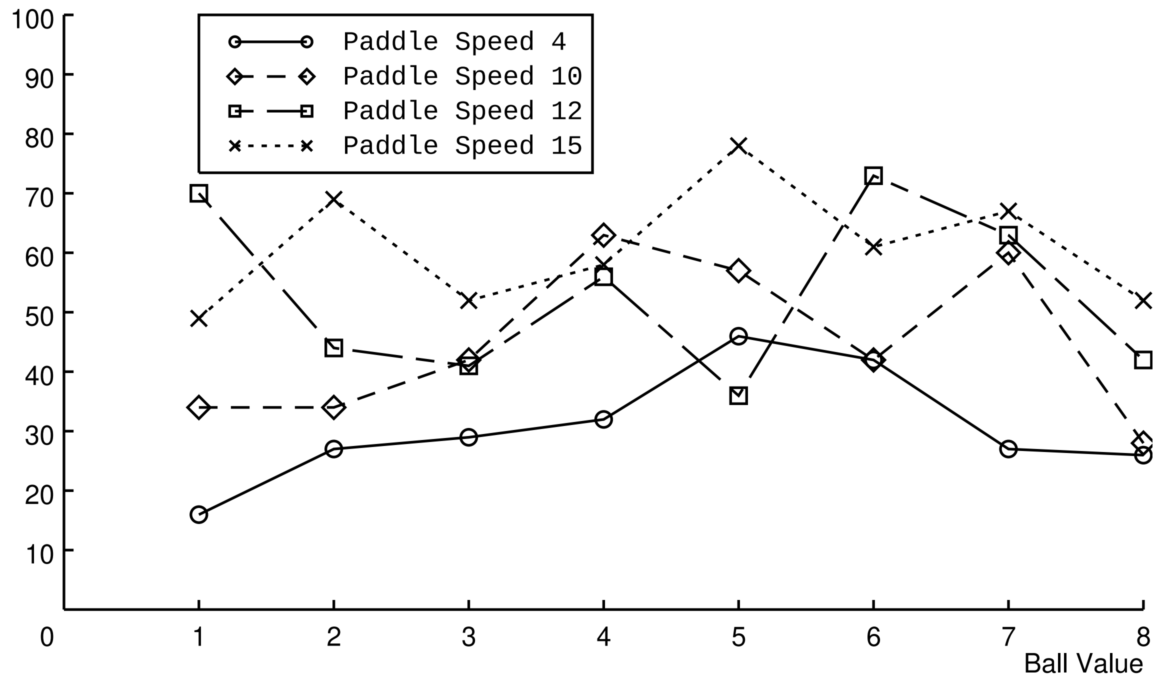

Figure 11: SPT Percent Caught

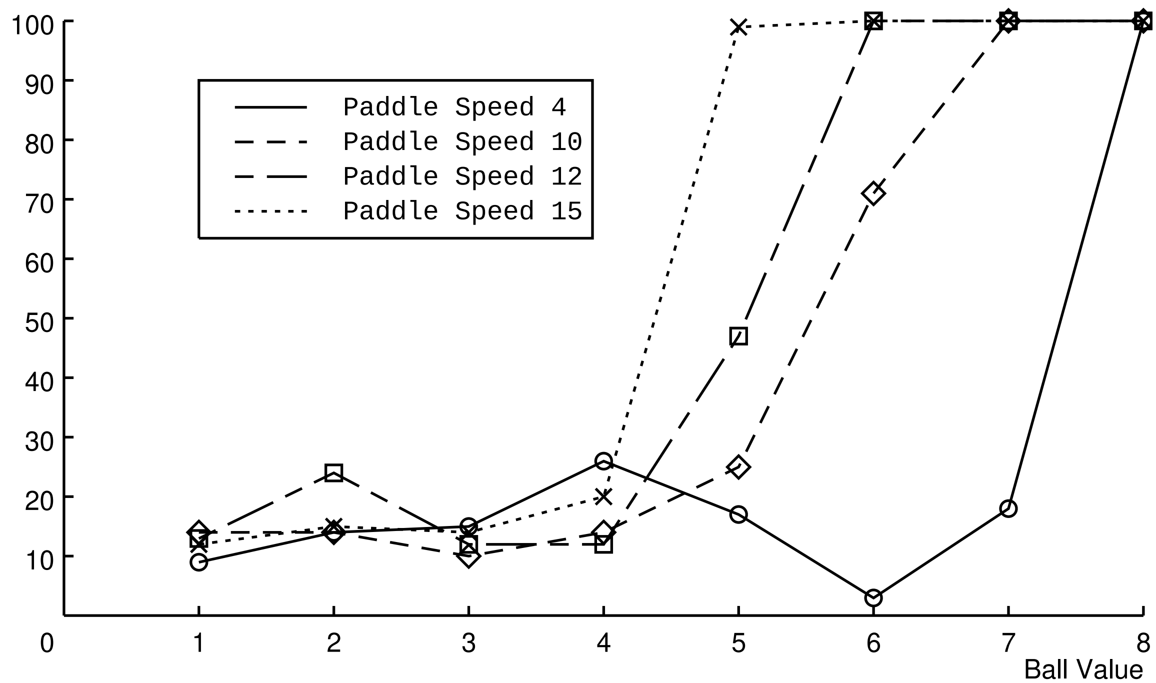

Figure 12: Static Priority Percent Caught

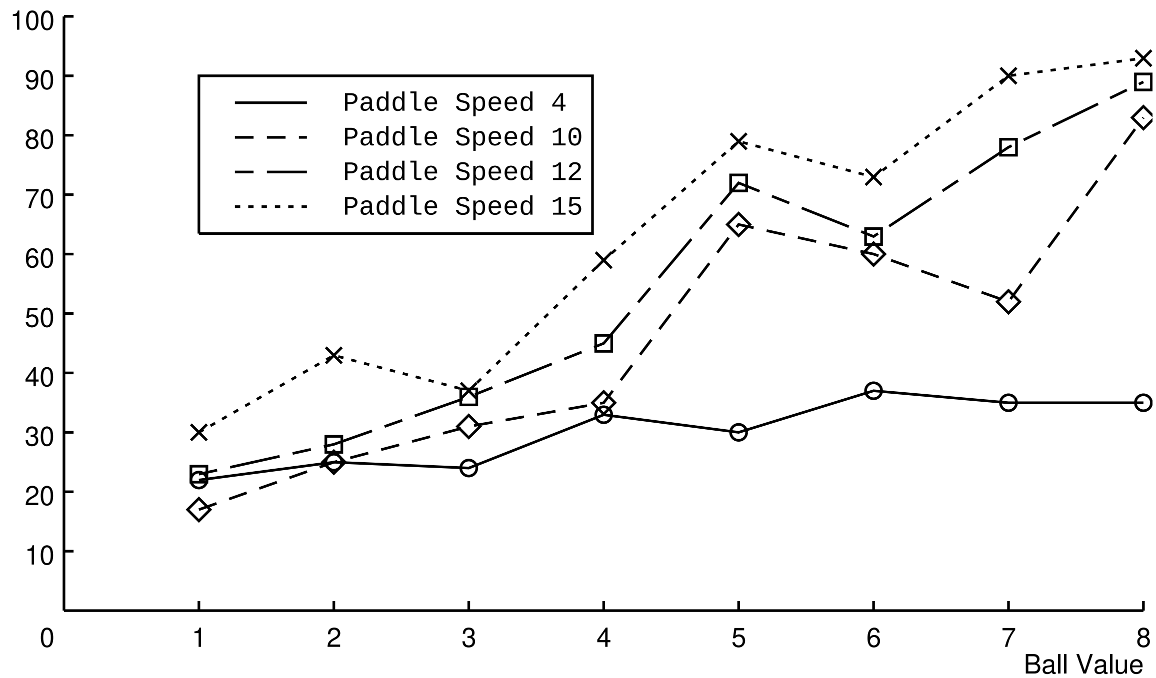

Figure 13: Best-Effort Percent Caught

Static Priority (Figure 12) has a very low catch rate for the small value balls, jumping to an almost perfect rate balls of higher values. Static Priority scheduling always runs the most important threads first. It therefore catches the high value balls (if it can), then attempts to catch the next highest value balls, etc. Increasing the paddle speed enables the application to catch more of the top value balls.

The characteristic curve for the Best-Effort policy (Figure 13) is smoother than those for the other policies. Like Static Priority, Best-Effort has a very good catching rate for the highest value balls. However, the rate of decrease between the high value balls and the lower value balls is more continuous. This behavior is a very important characteristic of the Best-Effort scheduler. Instead of concentrating only on the highest value balls, it tries to catch the combination of balls that maximizes the total value to the system. It may often be possible to schedule the successful capture of several balls of low value in place of a single one of higher value, and as a result provide better total value to the system. Compared to Static Priority's insistence on catching high valued balls to the exclusion of others, this selection can result in a considerable increase in the value obtained.

5.3.2 Meeting Application Time Constraints

One important method of comparing the schedulers is to examine how well each of them meets the thread deadlines, i.e. what fraction of the balls are successfully caught. The ratio of balls caught to the total number of potential catches is shown for each scheduler as a function of paddle speed in Figure 14. As expected, the ability of any scheduler to meet deadlines, and thus to catch balls, increases with paddle speed. It is interesting to note that the Deadline policy fares very poorly on this experiment. Since the system is heavily overloaded (all percentages are <100%), Deadline scheduling may waste significant time attempting to meet impossible time constraints. The SPT policy performs well since it concentrates on tasks which are easily accomplished. The Best-Effort policy performs as well as or better than any of the other policies because of its intelligent overload handling.

5.3.3 Maximizing Application Value

The total value metric is determined by dividing the value of the balls caught by the value that would have been obtained if all of the balls were always caught. Figure 15 illustrates the performance of each scheduling policy using this measure.

It is worthwhile to compare Figure 14 and Figure 15 (the two graphs are extracted from the same experimental data). The scheduling policies that ignore thread importance—Round Robin, Deadline, and SPT—catch almost the same percentage of balls as they do of value (their curves are practically identical in the two graphs). The two policies that consider thread importance—Static Priority and Best-Effort—catch more valuable balls and are shifted up by approximately 10% in the graph that indicates the percentage of the total value caught (Figure 15).

The policies that performed the worst using the value metric, Round Robin and Deadline, are also the schedulers that did the worst job of meeting time constraints. Clearly they achieve less value to the system simply because they catch fewer balls. The Shortest Processing Time scheduler achieves the next highest value to the system. It performs better than Round Robin and Deadline because it catches more balls, but it does worse than Static Priority and Best-Effort because it ignores value. Static Priority and Best-Effort are the most successful and are similar in performance for this particular set of experiments.

Static Priority and Best-Effort, are worthy of further consideration. One reason they perform well is that, unlike the other schedulers, value information plays an important part in scheduling decisions. One might wonder why Static Priority, which does not consider time constraints, is so successful at accruing value? The answer is found by examining how Static Priority distributes cycles to an arbitrary mix of threads. In overload, only the highest importance threads run (excluding all others) regardless of how likely their time constraints are to be met. In the short term, this may result in more dropped balls. If a thread's ability to meet an individual time constraint were independent of its past history, this behavior would result in a poor long-term performance as well. However, if expending cycles on a time constraint, even if the constraint is not satisfied, serves to improve the chances of meeting the next time constraint a thread establishes, then the cycles are not wasted and may benefit the system in the long term. This effect manifests itself in this application. If a scheduler concentrates solely on the highest value balls, some balls that could have been caught may be dropped. However, because they are receiving the majority of the processing cycles, the high value balls will tend to be the ones which are the easiest to catch. As a result, scheduling decisions which appear to be bad based on the available information may be good in the long term. Some applications do exhibit this type of "feedback;" however, the majority of tasks have a greater independence between activations.

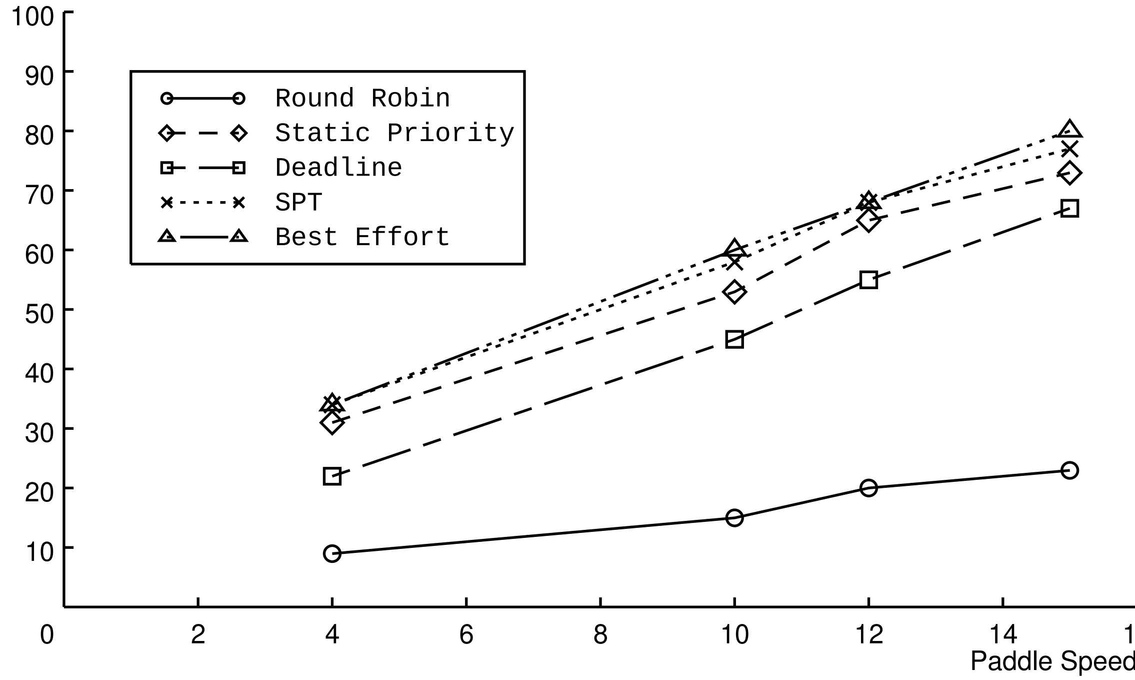

Figure 14: Percent of Balls Caught

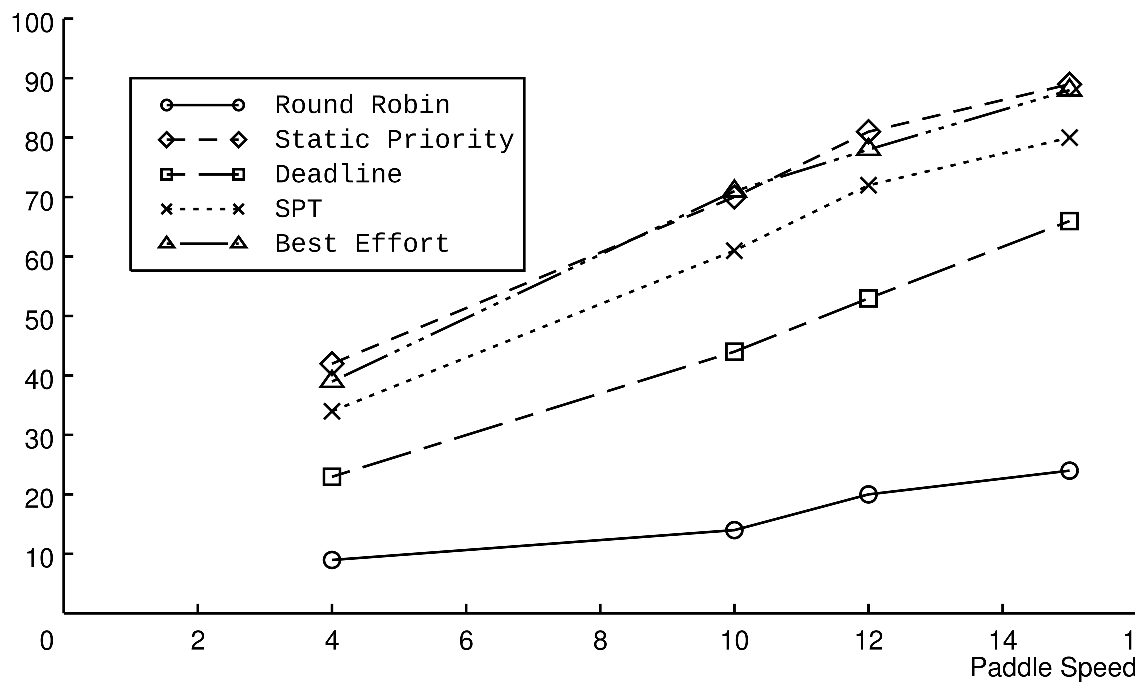

Figure 15: Percent of Value Caught

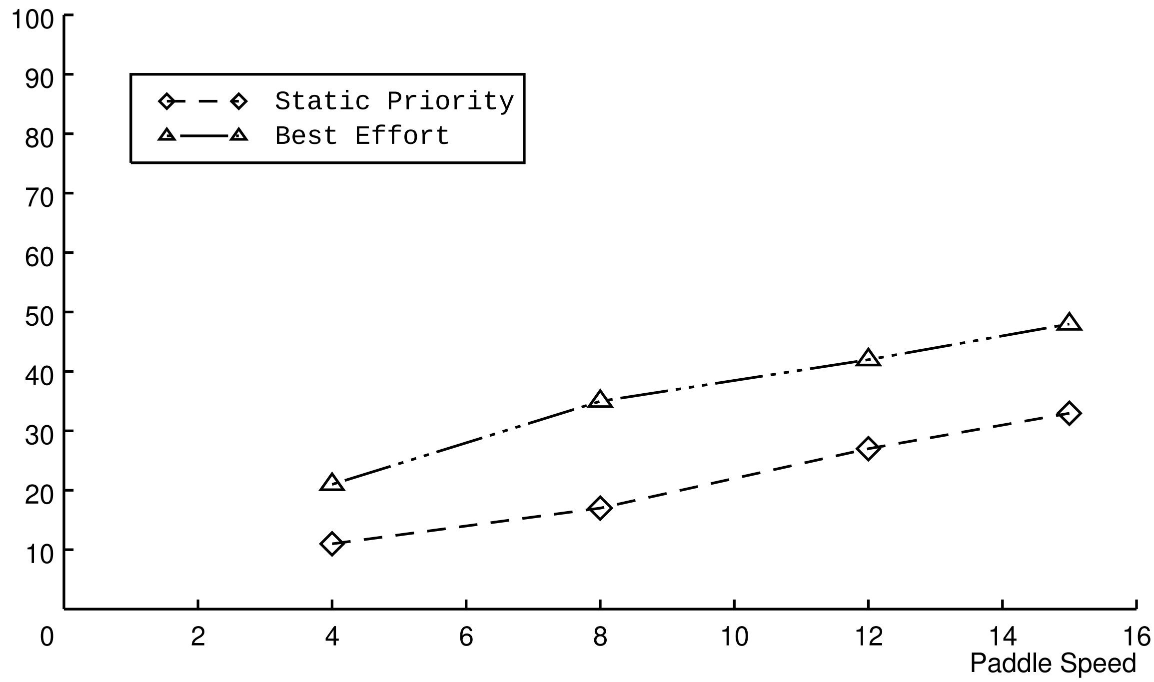

Figure 16: Percent of Value Caught with Random Replacement

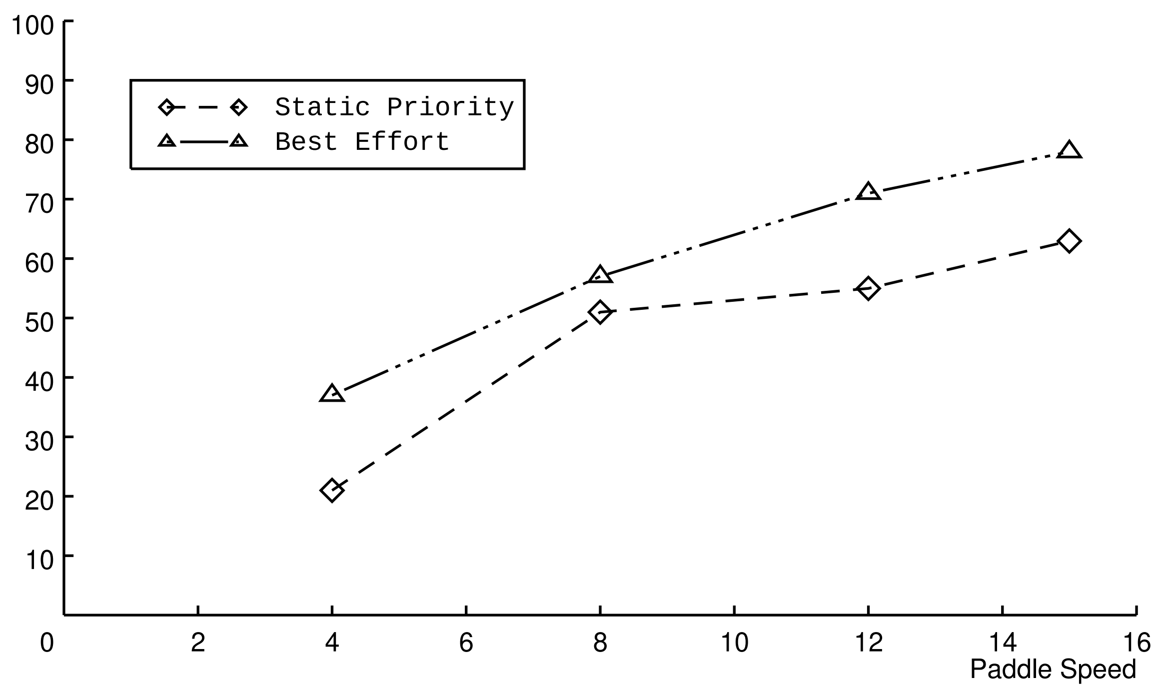

Figure 17: Percent of Value Caught with Smaller Value Variance

To study the behavior of independent tasks, a second series of experiments was performed. The application was altered so that after balls are caught or dropped they move to a random point in the air and begin falling again. Therefore, cycles spent on an unsuccessful catch is time poorly spent. A bad short-term decision cannot in general be converted into a good long-term one. The results of this modification are shown in Figure 16.

Another point to consider is whether the choice of values for the eight balls has an effect on the performance of the two schedulers that consider thread importance. The original set of eight balls valued one through eight was replaced by three balls of value six and five of value five. The results are shown in Figure 17. The comparative performance of the Best-Effort policy improves with this change in value distribution. The key to understanding this relative improvement in Best-Effort's performance is the fact that the balls with the highest value are no longer so useful to the system that focusing on them to the exclusion of others is profitable.

6. Conclusions

The best scheduler for a particular application would often be one designed precisely to match the requirements of the system. Such a scheduler could have complete knowledge of the application and could exploit application-specific information to achieve the best possible performance. For certain low-level systems where there are few different types of activities or where the events occurring in the system are completely predictable, custom schedulers may be feasible. As the variety of time constraints and frequency of unexpected events increases, however, it becomes more and more difficult to construct an application-specific scheduler that will operate correctly.

Since building a custom scheduler for each application is impractical, scheduling policies have been developed that utilize application-specified hints or requirements information to schedule activities in the way that will most benefit the application. In general, the more information given to the scheduler, the better the scheduling decisions can be. One consequence of using additional information is that scheduling decisions may become more complex. It is therefore necessary to balance the complexity of the scheduling operations with the resulting improvement in the application performance.

In the scheduling policies tested in this work, the amount of application-specified information used to make scheduling decisions varied from none (Round Robin) to a complete time-value function (Best-Effort). Predictably enough, it was the scheduler that used the most information about the application, Best-Effort, that showed the best behavior. The tests indicate that the extra computation time required in the Best-Effort algorithm was small enough to make the Best-Effort scheduling policy a good balance between performance and computational complexity.

References

[Jensen 75] Jensen, E. D.
Time-Value Functions for BMD Radar Scheduling.
Technical Report, Honeywell System and Research Center, June 1975.

[Locke 86] Locke, C. D.
Best-Effort Decision Making for Real-Time Scheduling.
Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, May 1986.

[Northcutt 87] Northcutt, J. D.
Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel.
Academic Press, Boston, 1987.

[Northcutt 88a] Northcutt, J. D.
The Alpha Operating System: Requirements and Rationale.
Archons Project Technical Report #88011, Department of Computer Science, Carnegie-Mellon University, January 1988.

[Northcutt 88b] Northcutt, J. D. and Clark, R. K.
The Alpha Operating System: Programming Model.
Archons Project Technical Report #88021, Department of Computer Science, Carnegie Mellon University, February, 1988.