Chapter 6

Experimental Evaluation

This chapter presents an experimental evaluation of the implementation of processor reserves in RT-Mach. The reservation mechanism was designed to support predictable behavior for real-time and multimedia applications, so the evaluation answers the questions: Can reserved programs achieve predictable behavior, and what is the price of predictability?

6.1 Overview

The experimental evaluation presented in this chapter answers two questions: Can reserved programs achieve predictable behavior, and what is the cost for predictability? These questions are addressed using synthetic benchmarks, real applications, and measurements of individual mechanisms. The chapter is divided into two main sections, one to address predictability and another to address scheduling costs.

The section on predictability shows that for a wide variety of task sets, real-time tasks exhibit predictable behavior and meet their timing constraints:

- Independent synthetic workload measurements show that for pure computations that have no interactions with other tasks, the reservation mechanism successfully guarantees timing constraints.
- Client/server experiments show that the reserve propagation mechanism helps guarantee the client's timing constraints, even when there are multiple clients with and without reservations.
- Results of experiments with the QTPlay QuickTime video player and the X Server show how this client/server pair is coordinated to meet the timing constraints of the video player, even when there are unreserved X clients competing for the attention of the X Server.
- Experiences with the mpeg_play decoder and the X Server demonstrate how an application can start with an inaccurate estimate of required com-

putation time and then adjust its reservation parameters to balance its usage requirements with the resource availability.

- Experience with a library-based network protocol software structure shows that protocol processing for real-time applications can be guaranteed using processor reserves.

The other main section of this chapter explores the scheduling costs of the reservation system. Two measurement techniques are used:

- A comparison of the system scheduling costs for periodic real-time programs that use reserves vs. periodic programs that do not use reserves shows that the cost varies, as expected, depending on the period of the program.
- Measurements of various internal operations such as reserve switch time, overrun timer handling, replenishment timer handling, and usage checkpoint operations provide a means of estimating scheduling cost for reserved task sets.

The measurements for most of these experiments were taken using RT-Mach version MK83j with UNIX server UX41. The mpeg_play and libsockets experiments used RT-Mach version MK83i, which does not different significantly from MK83j in the features used in the experiments. The hardware platform for the first three sets of experiments was a 90MHz Pentium with 16 MB RAM and an Alpha Logic STAT! timer card. The timer card has a 48-bit free-running clock with 1 μs resolution, and a 16-bit interrupting timer with 1 μs resolution. For the remaining experiments, the hardware platform was the same except the processor was a 486 DX2 instead of a Pentium. For easy reference, the chart below summarizes which platforms were used for which experiments.

Experiment	RT-Mach Version	UX Version	Processor
Independent task sets	MK83j	UX41	Pentium
Client/server task set	MK83j	UX41	Pentium
QTPlay/X Server	MK83j	UX41	Pentium
mpeg_play/X Server	MK83i	UX41	486
libsockets	MK83i	UX42	486
Aggregate scheduling costs	MK83j	UX41	486
Micro measurements	MK83j	UX41	486

Table 6-1: Summary of Testbed Platforms

In general, the switch from the 486 to the Pentium speeds up the compute-intensive applications by about 30%. Since the micro measurements involve kernel instruction streams that access external devices such as the clock/timer card, these measurements are not expected to change significantly using a Pentium processor.

In summarizing the results of many of the experiments, percentiles are used to specify dispersion. While running these experiments on a desktop computer connected to the normal departmental network, occasional anomalies occurred. 5-percentiles and 95-percentiles are used to indicate the range of the strong majority of measurements while ignoring the occasional anomaly. As defined in Jain's book [50], the 5-percentile is obtained by sorting a set of n observations and taking the [1 + n(.05)]th element in the sorted list (where [.] is used to indicate rounding to the nearest integer). The 95-percentile is the [1 + n(.95)]th element in the sorted list.

6.2 Predictability

What is meant by "predictability?" In the context of this work, a predictable application is one whose timing behavior can be determined from the application code, the resource reservations that it acquires, and its dependence on other programs. In particular, a predictable application that is not under-reserved will meet all of its timing constraints.

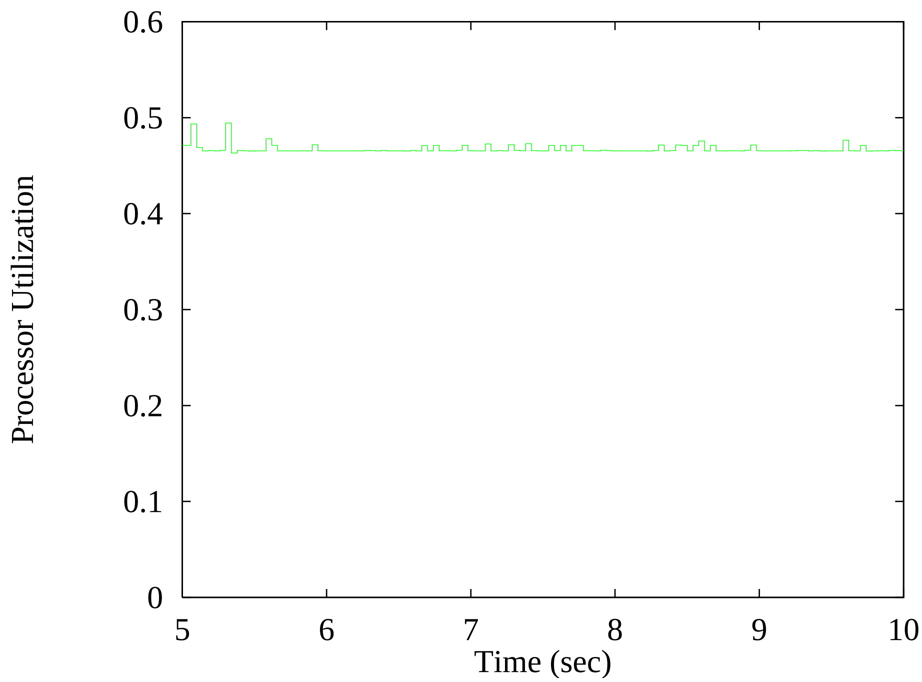

Figure 6-1: Compute-Bound Periodic Task with No Competition

As an example, consider a periodic application that computes for a fixed duration of time in each period and spends all of its computation time in a tight loop. Such an application should be able to allocate a reservation for its computation time, and it should exhibit the same behavior when it is executing concurrently with other activities as when it is executing in isolation. That is, it should be able to consume its reserved computation time in each period before the “deadline” at the end of the period.

Figure 6-1 shows the processor usage over time of a periodic application with a local computation and no competition for resources. The x-axis is time measured in seconds; it shows several seconds of usage information for the application. The y-axis is the normalized processor utilization of the application. Time on the x-axis is divided into intervals that correspond to the period of the application, and the processor time used during each period is measured and the utilization computed for the period. The utilization is plotted at that constant level for each period. For the duration of the test shown, he periodic application has an average utilization of 0.467. The distribution of measurements is very closely packed around this average with a 5-percentile of 0.465 and a 95-percentile of 0.473.

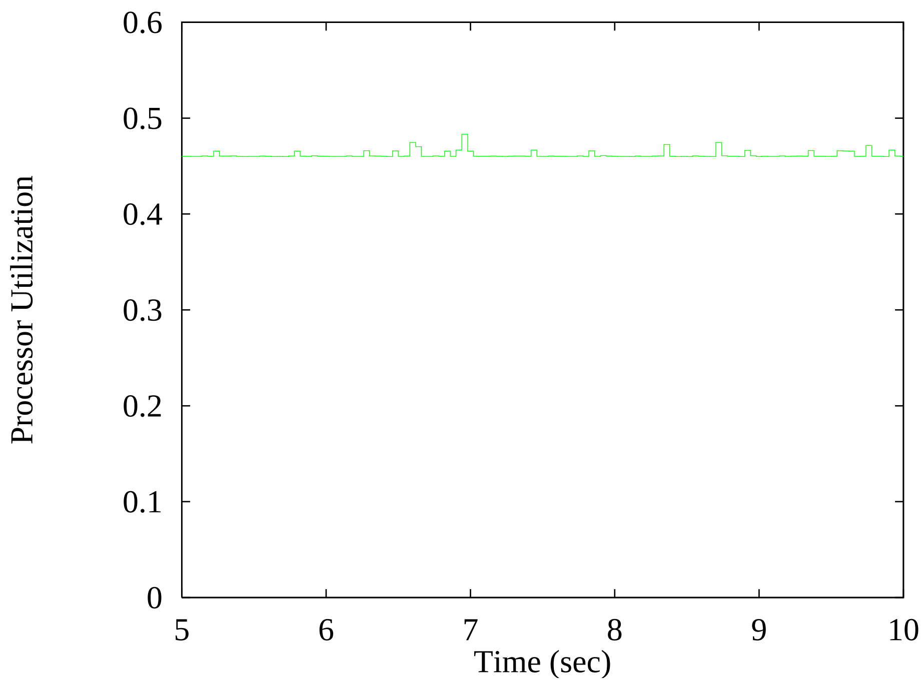

Figure 6-2: Compute-Bound Periodic Task with Competition

Figure 6-2 shows a similar graph of the processor usage of a periodic application that has a local computation but has competition for the processor from other programs (not shown). Even though there is competition, the reservation system ensures that the appropriate amount of processor time will be available to the application in each period. The application consumed an average of 0.462 of the processor in each period. As in the previous case, most of the measurements were very close to the average; the 5-percentile is 0.460 and the 95-percentile is 0.470.

6.2.1 Independent synthetic workloads

Independent synthetic workloads are used to test whether the reservation system can successfully provide access to reserved processor resources. The example above demonstrates that the reservation system can ensure predictable behavior for a periodic application running with competition from other activities, and one of the experiments described below shows that multiple independent reserved applications can achieve predictable behavior, even with competing unreserved activities. Two additional experiments show that the reservation guarantee is independent of the number of competing activities, regardless of whether the competitors are reserved or unreserved.

6.2.1.1 Methodology

These experiments were run using two software tools developed for performance evaluation. A configuration manager parses the specification of a task set with timing parameters and reservation parameters and then creates programs with the appropriate parameters. Several different kinds of programs that exhibit different kinds of behavior can be specified in the task set. Each of these programs takes a start time, a duration to compute, a thread period, a computation time to reserve, and a reservation period. Two programs are used in these experiments:

- `arith` - Creates a periodic thread that executes in a tight loop for some duration of time in each period.
- `monitor` - Records the usage charged to reserves in the experiment. This program has a reservation of its own to enable it to run even when there are many reserved programs in the experiment.

A usage monitor is usually included in the task set to take usage measurements for all of the programs created by the configuration manager. The monitor buffers the measurements during the course of the experiment and then formats the data and writes them to disk after the experiment is completed. The data are then graphed.

Experiment 1 is designed to show that reserved activities are able to execute their periodic computations within their time constraints, even with competing unreserved activities. Even if the reservation parameters have different computation times and different periods the timing constraints of the reserved activities will be satisfied.

Table 6-2 shows the programs used in Experiment 1 along with the number of instances of each program, the timing parameters, and the reservation parameters. In this experiment there were 3 `arith` programs that were reserved with different timing and reservation parameters. One had a reservation of 5 ms of every 20 ms, the second had a reservation of 14 ms every 40 ms, and the third a reservation of 8 ms every 50 ms. The reservation is set slightly higher (1 or 2 ms) than the computation time that would be consumed by the program in isolation. This accommodates variation in the computation time due to cache effects and context switches. In addition to those three, there were 5 `arith` programs running in infinite loops with no reservations; these provide compute-bound competition for the reserved activities. And finally, the experiment included a `monitor` program to collect usage numbers throughout the duration of the test. This monitor had 2 ms reserved of every 20 ms.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
`arith`	1	4 ms	20 ms	5 ms	20 ms
`arith`	1	12 ms	40 ms	14 ms	40 ms
`arith`	1	6 ms	50 ms	8 ms	50 ms
`arith`	5	infinite loop	N/A	0 ms	40 ms
`monitor`	1	N/A	20 ms	2 ms	20 ms

Table 6-2: Experiment 1 Parameters

The other two experiments measure the sensitivity of reserved applications to competition. Both experiments consist of eight series of tests. Each test has a reserved `arith` program whose measurements are the focus of the test. The series differ in that the reserved `arith` program increases in reserved utilization in each series. Each series itself consists of a sequence of tests with an increasing number of competitors. In each series of Experiment 2, the one reserved `arith` program competes with an increasing number of unreserved `arith` programs. For each test, the 5-percentile and 95-percentile for the resource usage measured in each reservation period is reported. The parameters for an example task set in Series 1 of this experiment appear in Table 6-3.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
`arith`	1	3 ms	40 ms	4 ms	40 ms
`arith`	2	infinite loop	N/A	0 ms	40 ms
`monitor`	1	N/A	20 ms	2 ms	20 ms

Table 6-3: Example Parameters for Experiment 2

The task set that appears in Table 6-3 has one reserved `arith` program that computes 3 ms in every 40 ms and has a reservation of 4 ms for every 40 ms. It also has two unreserved `arith` programs and a `monitor` program. Other tests in the Series 1 have unreserved competitors ranging in number from 0 to 9 competitors. This task set is designed to show that regardless of what the reserved utilization is and regardless of how many competitors there are (ranging from 0 to 9 compute-bound, unreserved competitors), a reserved activity will always be able to get its reserved allocation.

Experiment 3 is like Experiment 2 except that the competitors are reserved instead of unreserved. For convenience, the competitors all have identical reservations, so the number of competitors for a reserved activity is limited to the number of competitors that can be accepted by the admission control policy.

The tests of Experiment 3 are organized into 8 series with 10 tests, just as in Experiment 2. Again, the series differ in that the reserved `arith` program that is observed varies in its reserved utilization, and within each series, the number of competitors ranges from 0 to the highest number that can pass admission control along with the observed program. Variation in the usage of the measured reserved program is again characterized by the 5-percentile and 95-percentile. The parameters for an example test in this experiment appear below.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
arith	1	3 ms	40 ms	4 ms	40 ms
arith	2	2 ms	30 ms	3 ms	30 ms
monitor	1	N/A	20 ms	2 ms	20 ms

Table 6-4: Example Parameters for Experiment 3

The task set in Table 6-4 has one `arith` program with computation duration 3 ms and a period of 40 ms. The reservation given to this program is 4 ms every 40 ms. The table lists two other reserved `arith` programs with 2 ms computation time and 30 ms period, and these both have reservations of 3 ms every 30 ms. This is a test from Series 1 of Experiment 3, and other tests in this series have the different numbers of competing reserved programs. The number of competitors for Series 1 ranges from zero to seven, but the number of competitors for Series 8 is zero since no competitors could be admitted once the primary reserved program and the reserved monitor pass admission control. The purpose of this task set is to demonstrate that regardless of what the reserved utilization of the primary reserved program and regardless of the number of competitors, the primary reserved activity will get its reserved allocation virtually all of the time.

6.2.1.2 Results

The results from Experiment 1 demonstrate that multiple reserved programs meet there timing constraints, despite the competition between the reserved activities and competition from unreserved activities. Figure 6-3 shows a graph of the behavior of the three reserved programs in Experiment 1, leaving out the usage measurements of the competing unreserved activities. These usage measurements are in the same format as the example case described earlier: the x-axis is time in seconds for the test, and the y-axis is processor utilization. The usage for each reservation period for each reserved program is computed and plotted on the graph, yielding three functions of utilization over the duration of the test.

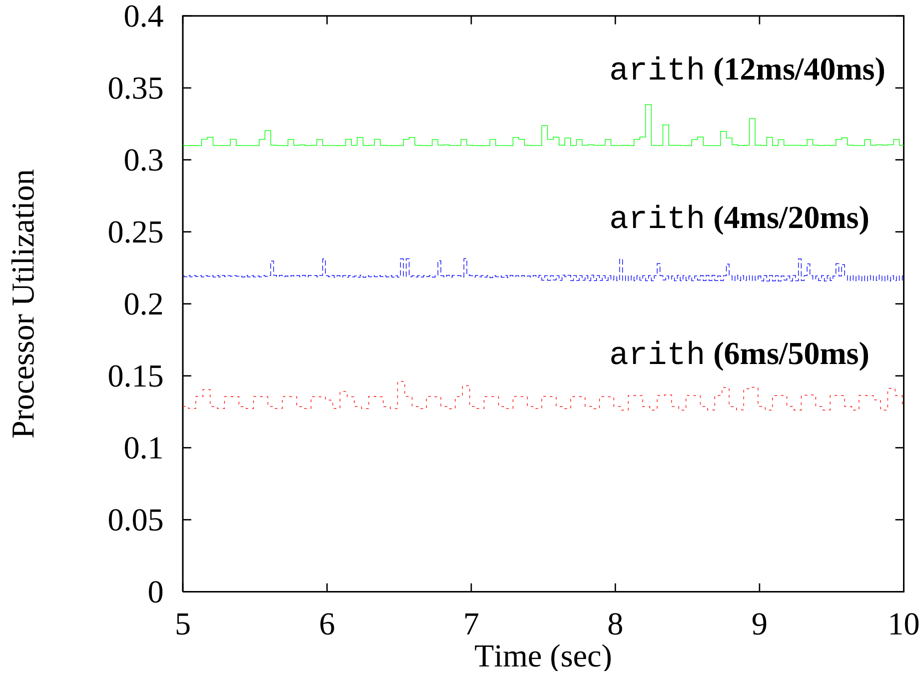

Figure 6-3: Experiment 1 Results

Each of the three reserved programs sustains a fairly constant utilization level throughout the entire test, in spite of the competition from reserved and unreserved activities. The reserved program with a computation time of 4 ms every 20 ms gets a fairly constant utilization with an average of 0.219 (context switching overheads and cache effects push the measured usage higher than what it would be in a quiescent system). The 5-percentile is 0.216 and the 95-percentile is 0.227, indicating that very few measurements fall far from the average. The reserved program computing 12 ms every 40 ms gets an average utilization of 0.312. It gets a 5-percentile of 0.310 and a 95-percentile of 0.320, so there is clearly very little variation in the utilization across periods. The program computing 6 ms every 50 ms gets an average utilization of 0.132 across the periods shown above. The 5-percentile is 0.126 and the 95-percentile is 0.141. Thus Experiment 1 shows that the reservation system can guarantee the timing constraints for multiple reserved programs even when there is competition from unreserved activities.

The results from Experiment 2, illustrated in Figure 6-4, show that the timing behavior of a reserved program is not affected by the number of unreserved competitors, regardless of the utilization of the reserved program. The graph in Figure 6-4 has the number of competitors on the x-axis and processor utilization on the y-axis. The data from the eight series are plotted as functions on the graph. For example, the function for Series 1 starts with the average utilization for the test that has zero competitors. The function then continues to the average utilization for the test in that series that has 1 competitor and so on up to the average utilization for the test with nine competitors. At each point where the average utilization is plotted, there is also a range that gives the 5-percentile and 95-percentile to indicate the variation in the behavior of the reserved program on that test. The rest of the functions are similar. This graph shows that for each series, the average processor utilization is nearly constant, regardless of the number of competitors. Furthermore, the variation given for each measurement is quite small, indicating that for the vast majority of reservation periods, the reserved program is able to meet its timing constraints.

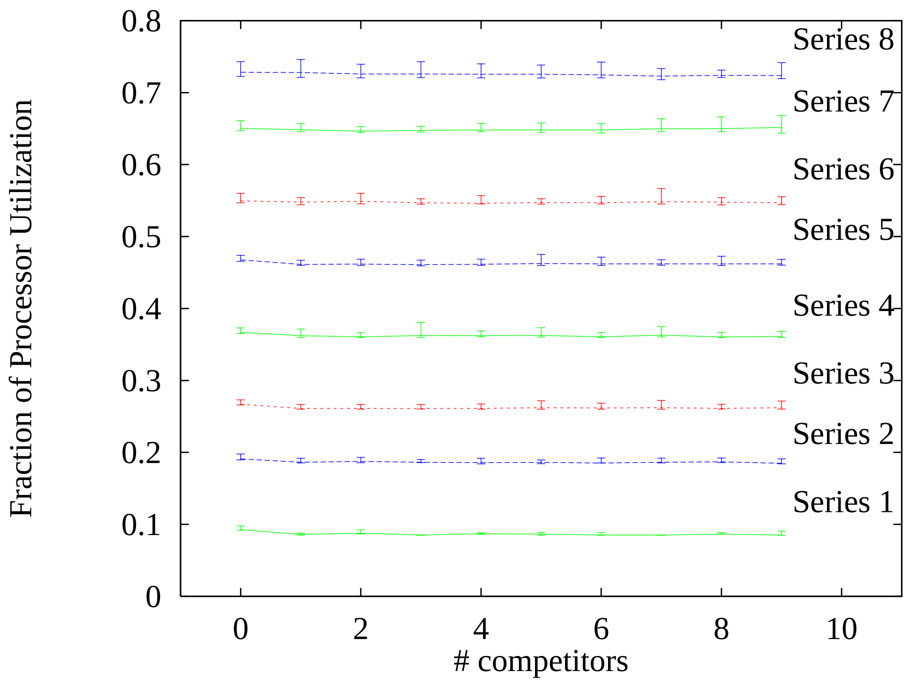

Figure 6-4: Experiment 2 Results

The results for Experiment 3 are presented in Figure 6-5. These results show that the behavior of a reserved program does not depend on the number of competing reserved activities, regardless of the utilization of the reserved program of interest. The graph for Experiment 3 is much like the graph for Experiment 2. The x-axis is the number of competing programs, and the y-axis is processor utilization. There are data from the same kinds of series, and the plot of average utilization as a function of number of competitors is the same. Each plotted point has a 5-percentile and 95-percentile range to indicate the variation. Since each of the competitors must pass the admission control policy, the number of competitors becomes more limited as the utilization of the measured reserved activity gets larger. So the maximum number of competitors for Series 1 is seven and for Series 8, no competitors can pass admission control after the measured reserved activity and the monitor do. These results show that the average processor utilization for each series is nearly constant, i.e. it does not depend on number of competitors. The variation in processor utilization is very small, indicating that the reservations are available to allow the reserved activity to satisfy its timing constraints. This is true regardless of the processor utilization of the measured reserved program.

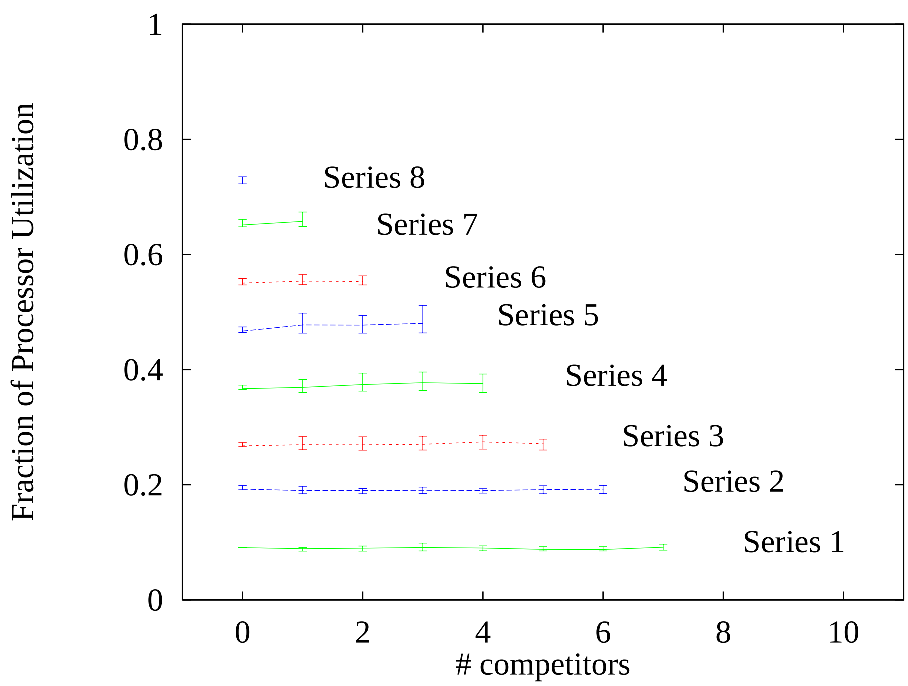

Figure 6-5: Experiment 3 Results

6.2.1.3 Analysis

These three experiments show that for cases where periodic threads allocate reserves for their computations, the reservation system is able to ensure that the reserved time is available as promised. The reserved time is available even when there are multiple reserved activities and unreserved competitors as in Experiment 1. Experiment 2 showed that a reserved activity is assured of being able to use its reserved time regardless of the number of unreserved competitors and regardless of whether the reservation is for a small computation time or a large computation time. Experiment 3 demonstrated that a reserved activity will get its reserved time regardless of the number of reserved competitors it has, and this is also true whether the reserved activity has a small amount of time reserved or a large amount of time.

Two issues are highlighted by these experiments. One is that the computation time for the reserved programs was always less than the reserved computation time by 1 to 2 milliseconds. The computations that the programs will execute are based on arithmetic computations that were timed on a quiescent system with only the timing program running. The synthetic workload was tuned in this environment. When these workloads are executed with other activities, there are additional overheads that are not included in the task set specifications. These overheads include:

- context switch times and cache effects, which are likely to increase as the task set size increases,
- periodic thread overhead associated with periodically releasing and resetting the computation,
- and interference from interrupts.

The reserved computation time is set to be 1 to 2 milliseconds larger than the pure computation amount to accommodate these overheads.

The second issue is that in some rare cases, a series of interrupts or a large critical region in the kernel may preempt a program and cause it to miss its reserved time and subsequently miss its deadline for the period. To mask these rare instances in Experiments 2 and 3, the 5-percentile and 95-percentile are given. This shows that for the vast majority of reservation periods for these threads, the usage observed is that which is expected based on the reservation.

6.2.2 Client/server synthetic workloads

Most interesting applications are not independent, so it is important to consider experiments that characterize the effect of interactions such as client/server relationships in reserved applications. The experiments described below show that reserved activities can achieve predictable behavior, even when the activities involve coordination between clients and servers.

6.2.2.1 Methodology

The following experiments use the same kind of software environment as described in Section 6.2.1. There is a configuration manager that reads a specification of a task set and then creates the programs for the task set. In addition to the `arith` and `monitor` programs described above, these experiments use the following programs:

- `tsclient` - Creates a periodic thread that invokes a server to compute for some duration of time specified in the invocation. The invocation is performed using the regular Mach IPC mechanism.
- `tsserver` - Services requests sent in from instantiations of the `tsclient` program.
- `rclient` - Creates a periodic thread that invokes a server to compute for some duration of time, but unlike the `tsclient`, the `rclient` uses RT-IPC instead of regular Mach IPC, and the `rclient` sends a reserve to the server so that it can charge the computation time to the client's reserve.

- `rserver` - Services requests sent from instantiations of the `rclient` program. Uses RT-IPC and charges the computation requested by a client to the client's reserve, which is passed as an argument with the invocation.

Experiment 4 is designed to show the processor usage of a client/server pair that has no competition from other programs; this is the base case, showing the desired behavior for the client and server. It uses a task set with an instance of the `tsclient` program using Mach IPC to periodically invoke an instance of `tsserver` to perform a computation. A monitor records the usage for later analysis. In this case, the client sends the computation time amount (to be consumed in a tight loop) to the server. And the server computes for that amount of time and returns a result. The parameters for the programs in this task set are given in Table 6-5. The client is periodic and has a reservation associated with it. As long as the computation time requested by the client is smaller than the period, the server with no competition should be able to finish the computation by the end of the period, yielding a fairly constant utilization over time.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
`tsclient`	1	10 ms	40 ms	10 ms	40 ms
`tsserver`	1	infinite loop	N/A	0 ms	40 ms
`monitor`	1	N/A	20 ms	1 ms	20 ms

Table 6-5: Experiment 4 Parameters

In Experiment 5, the task set includes competition from unreserved programs as well as the `tsclient`, `tsserver`, and `monitor`. This experiment is meant to show how competition for the processor from unreserved activity can interfere with the coordinated activity of a client and server using a typical IPC mechanism.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
`rclient`	1	8 ms	40 ms	10 ms	40 ms
`rserver`	1	N/A	N/A	0 ms	40 ms
`arith`	5	infinite loop	N/A	0 ms	40 ms
`monitor`	1	N/A	20 ms	2 ms	20 ms

Table 6-6: Experiment 6 Parameters

Experiment 6 is designed to determine whether a client/server pair, using an IPC mechanism integrated with the reservation system in terms of queueing and scheduling, can sustain a predictable, coordinated activity even with competition for the processor. Table 6-6 shows the task set specification for Experiment 6. The rclient has a processor reservation, and the `rclient` and `rserver` communicate using RT-IPC. The competition for the processor comes from five `arith` programs which are unreserved.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
`rclient1`	1	8 ms	40 ms	10 ms	40 ms
`rclient2`	1	8 ms	50 ms	10 ms	50 ms
`rclient3`	1	8 ms	60 ms	10 ms	60 ms
`rserver`	1	N/A	N/A	0 ms	40 ms
`arith`	5	infinite loop	N/A	0 ms	40 ms
`monitor`	1	N/A	20 ms	2 ms	20 ms

Table 6-7: Experiment 7 Parameters

Experiment 7 is intended to show whether several reserved clients can execute in a manner that satisfies their timing constraints when using the same server and competing with unreserved, compute-bound programs. The task set, shown in Table 6-7 shows three reserved instances of the `rclient` program with different reservation parameters. There is also an instance of the `rserver` program. The competition comes from five instances of the `arith` program which are unreserved, and there is a reserved `monitor` program.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
`rclient`	1	8 ms	40 ms	0 ms	40 ms
`rclient`	1	8 ms	50 ms	10 ms	50 ms
`rclient`	1	8 ms	60 ms	10 ms	60 ms
`rserver`	1	N/A	N/A	0 ms	20 ms
`arith`	5	infinite loop	N/A	0 ms	40 ms
`monitor`	1	N/A	20 ms	2 ms	20 ms

Table 6-8: Experiment 8 Parameters

Finally, Experiment 8 is designed to determine whether reserved clients can meet their timing constraints if there is an unreserved client that is using the same server. As shown in Table 6-8, the task set for Experiment 8 contains one `rclient` program with no reservation and two `rclient` programs with reservations. There is an `rserver` and five competing `arith` programs which are unreserved. A `monitor` is also included in the task set.

6.2.2.2 Results

The results from Experiment 4, shown in Figure 6-6, illustrate the processor usage pattern of the periodic client and its server. The x-axis is time over the duration of the test measured in seconds, and the y-axis is processor utilization. The usage measurements for both the client and the server are taken from the corresponding reserves. The client has a reserve that it charges for its own computation, and the server has a reserve that it charges against when performing an operation for a client. For each of these reserves, the `monitor` records the computation time used in each reservation period. Those computation times are then normalized with respect to the length of the corresponding reservation periods and plotted as constant for the duration of each reservation period.

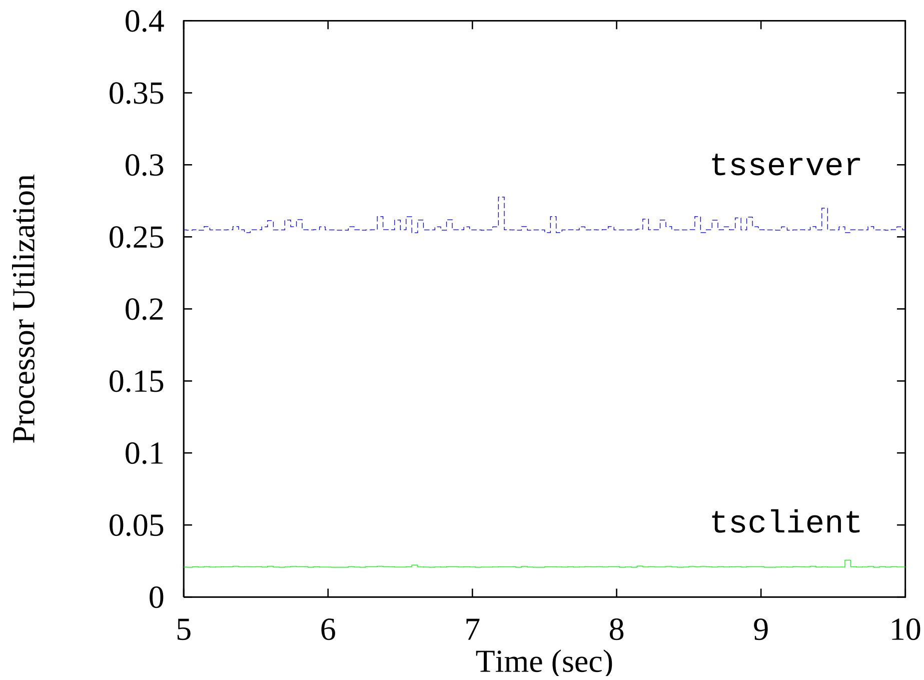

Figure 6-6: Experiment 4 Results

Figure 6-6 shows that the processor utilization charged to the server's reserve is fairly constant over the duration of the test. The average is 0.256 with a 5-percentile of 0.255 and a 95-percentile of 0.264 for the measurements graphed in the figure. The normalized charges to the client reserve average only 0.0210; the 5-percentile is 0.0207 and the 95-percentile is 0.0213. In this setup, the server is doing most of the work while the client does no work other than sending off requests and receiving replies. Since there is no competition and the client makes the same request in every period, the utilization is fairly constant over the duration of the test.

Figure 6-7 shows the results of Experiment 5 where the same client/server pair has competition for the processor from unreserved activities. As before, the x-axis is time measured in seconds, and the y-axis is processor utilization. In this case, the client and server do not have constant utilization numbers over each reservation period. For the measurements shown in the graph, the server has an average utilization of 0.191, which is significantly lower than the desired level. The 5-percentile for the server is 0, and the 95-percentile is 0.262. The client has an average utilization of 0.0117 across the periods shown in the graph; its 5-percentile is 0 and its 95-percentile is 0.0158. With the client and the server recording 0 utilization in a significant number of periods in a row, it is clear that the client/server combination is not achieving the desired behavior.

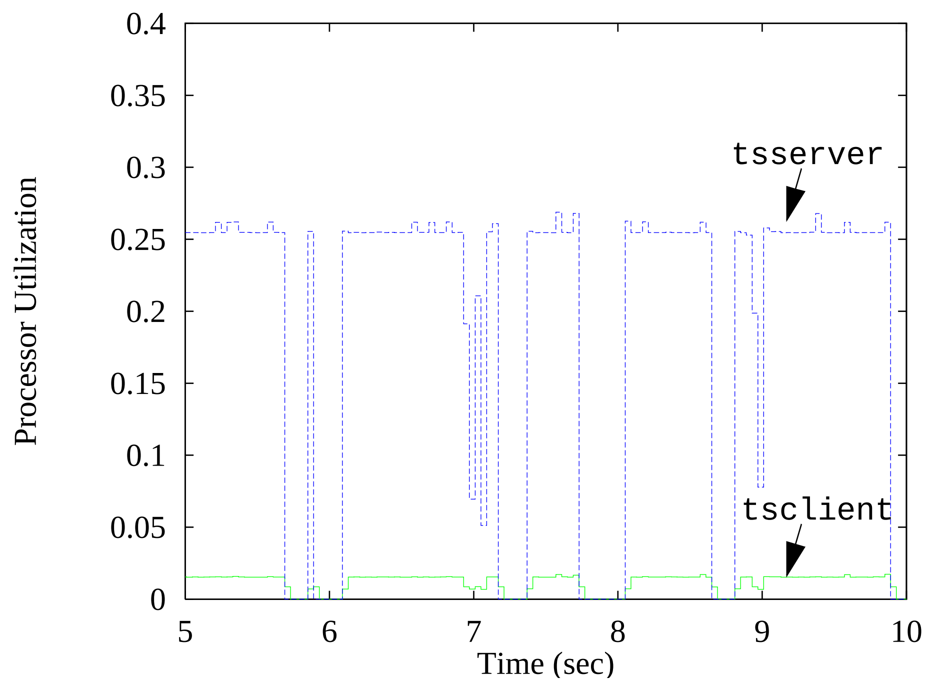

Figure 6-7: Experiment 5 Results

The competition from unreserved programs interferes with the execution of the server, and completely locks out the server for up to 100 to 200 ms at a time. During these periods, there is no usage recorded by either the server or the client, since the client cannot make progress without the server making progress. The usage for both the client and the server falls to zero for several reservation periods. This kind of behavior is clearly undesirable since many instances of the computation cannot take place, and the deadline is missed each time.

The results of Experiment 6 (Figure 6-8) show that when a client and server use an IPC mechanism that is integrated with the reservation scheduling policy, in this case a version of RT-IPC extended to work with the reservation scheduling policy, the combined client/server activity is quite predictable. The RT-IPC mechanism propagates the client's reserve to the server and supports a server that charges the computation time of each client to the client's reserve. So most of the computation in this experiment is being charged to the client (as it should be) instead of to the server (as in the previous case). The utilization charged to the client's reserve averages 0.223 with a 5-percentile of 0.222 and a 95-percentile of 0.228. The server utilization for the graphed intervals is 0.0114 on average; the 5-percentile is 0.0112 and the 95-percentile is 0.0117. These numbers indicate very predictable performance for these programs over time.

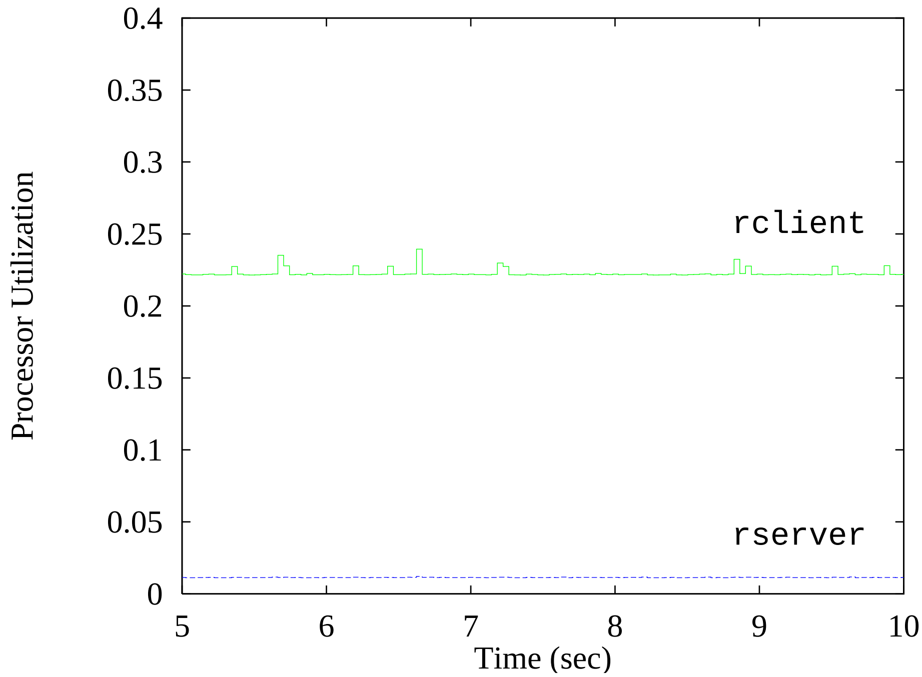

Figure 6-8: Experiment 6 Results

Figure 6-9 shows the results from Experiment 7. These results show that even when several reserved clients are using the same server, they can all meet their periodic timing constraints (subject to the admission control policy). This is true in spite of the presence of competition from unreserved `arith` programs.

Experiment 7 shows the usage charged to the reserves of the three reserved clients. The client with the 8ms/40ms synthetic computation has an average utilization of 0.223 with a 5-percentile of 0.221 and a 95-percentile of 0.232. The client with the 8ms/50ms computation has an average utilization of 0.178, a 5-percentile of 0.176 and a 95-percentile of 0.184. And the client with the 8ms/60ms computation gets an average utilization of 0.150; the 5-percentile is 0.147 and the 95-percentile is 0.157. These numbers indicate fairly tight distributions around the averages for these applications, even though they are competing for the server and even though there are additional unreserved programs competing for the processor as well. The servers computation time seems erratic, and there are two reasons: it has a small reservation period (which just determines the usage measurement period), and its clients all have different periods and request different computations at different rates.

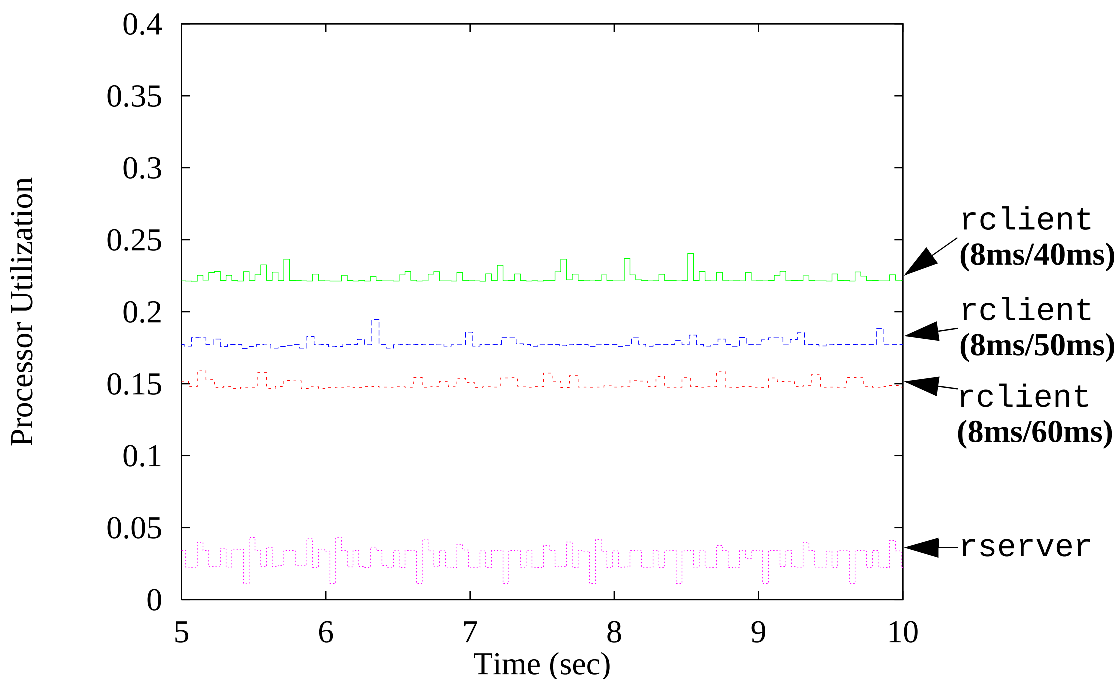

Figure 6-9: Experiment 7 Results

Finally, the results of Experiment 8 appear in Figure 6-10. These results show that in the case where reserved clients compete with an unreserved client for a single server, the reserved clients are still able to satisfy their timing constraints.

The reserved client with the 8ms/50ms computation has an average utilization of 0.181. It has a 5-percentile of 0.176 and a 95-percentile of 0.187. The reserved client with the 8ms/60ms computation has an average utilization of 0.151 with a 5-percentile of 0.148 and a 95-percentile of 0.158. Thus, these reserved programs are able to get the processor time they have reserved. As the graph shows, the unreserved client manages to complete its computation during some of its periods, but not in others. So the usage function goes back and forth between getting about 0.22 utilization in the periods where the computation is completed and getting 0 utilization in the periods where it does not get to complete the computation. The average utilization for this unreserved client is 0.131; the 5-percentile is 0.0059 and the 95-percentile is 0.222. This of course confirms that the dispersion of the utilization measurements for this unreserved client is large.

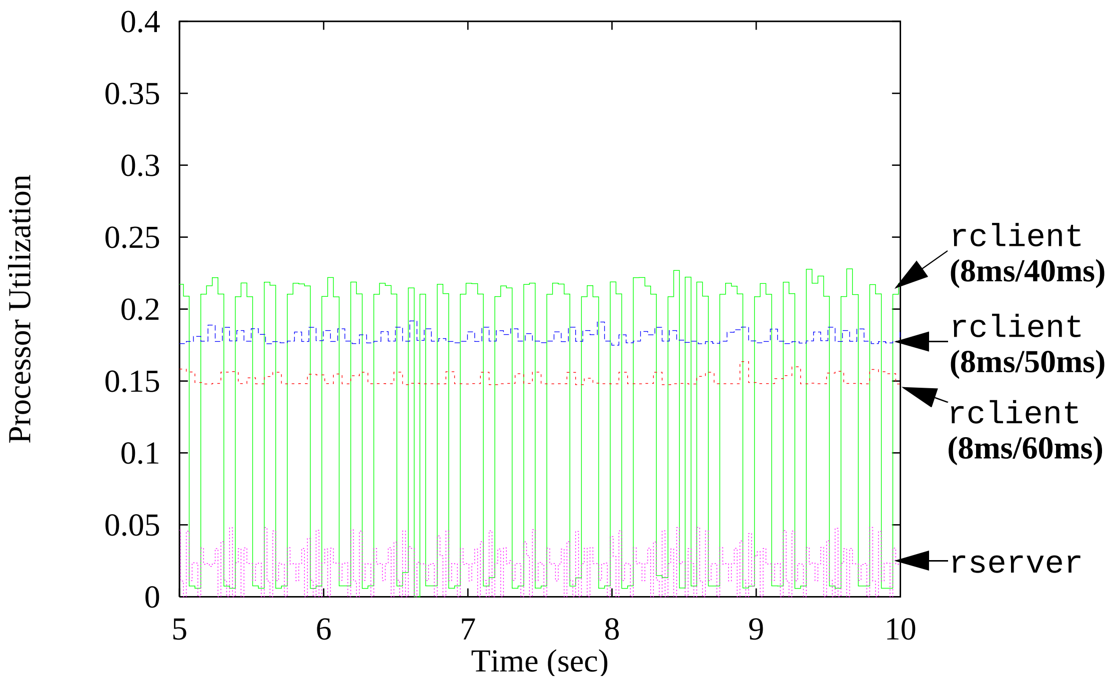

Figure 6-10: Experiment 8 Results

6.2.2.3 Analysis

The experiments with unreserved and reserved client/server pairs demonstrate the importance of doing reserve propagation properly between client and server when the server is designed to use the client's reserve. Experiment 4 shows the baseline behavior for the client/server pair with a periodic client driving the timing of the activity.

Experiment 5 demonstrates the problem that can occur when the reserve propagation is not handled properly. In this experiment, the client allocates a reserve and passes it to the server, which then uses it to charge the client's service time. However, this client/server pair does not use the "priority" inheritance mechanism to ensure that the server takes on the "priority" of the client as soon as the RPC is enqueued in the servers input queue. With competition from unreserved activities, this lack of "priority" inheritance results in many missed deadlines for the client/server activity.

The results of Experiment 6 show that the proper periodic behavior of the client/server pair can be restored by using the "priority" inheritance mechanism. Priority inheritance makes sure that the competing unreserved activities do not interfere with the server as it attempts to read the request from its input queue and switch to the client's reserve.

The last two experiments demonstrate that the reserve propagation mechanism, which includes "priority" inheritance and the server binding to the client's reserve, works properly even when there are multiple reserved clients or there are unreserved clients in addition to reserved clients. Experiment 7 demonstrates that with three reserved clients (all with different computation times and timing constraints), the server can service them all in time to make their deadlines. This is true even though there is competition from unreserved activities, which can exploit any lapses in an incorrectly implemented reserve propagation mechanism and cause delays in the client/server activities.

Experiment 8 shows that when two reserved clients and one unreserved client share a server in an environment with competition from other unreserved activities, the reserved clients will always meet their deadlines. In this case the reserved clients meet their deadlines even though other unreserved competitors sometimes delay the unreserved client. This experiment further tests the integrity of the reserve propagation mechanism by making sure that the "priority" of the unreserved client is not propagated to the server at the wrong time, causing the server to appear unreserved and resulting in interference from the unreserved competitors.

6.2.3 QTPlay/X Server

Experiments using task sets with synthetic workloads provide evidence that the reservation system can support the predictable execution of real-time programs. However, experiments with real applications that use the reservation system to meet timing constraints provide stronger evidence of the usefulness of the reservation system in real-world situations. The experiments described in this section use a video player that has been modified to use processor reserves and a version of the X Server that has been modified to support reserves.

6.2.3.1 Methodology

These experiments use the QuickTime video player, called `QTPlay`, and the reserved X Server, both of which were described in the previous chapter. Since these programs are described in detail elsewhere, the description here is brief.

The `QTPlay` application prefetches a short video clip into main memory and repeatedly displays that clip to avoid interaction with the file system and disk (which are not reserved in this system) during the experiments. It allocates a reserve during initialization based on command-line arguments and then starts playing the video. For each frame, the player records in a buffer the start time and end time for the frame processing, and at the end of the experiment these data are written to a file on the disk for later analysis.

When `QTPlay` connects to the X Server, it passes a reference to its reserve for the X Server to use when performing frame display operations. The X Server was modified to order requests based on reservation information and to charge the computation time for each operation to the appropriate client's reserve.

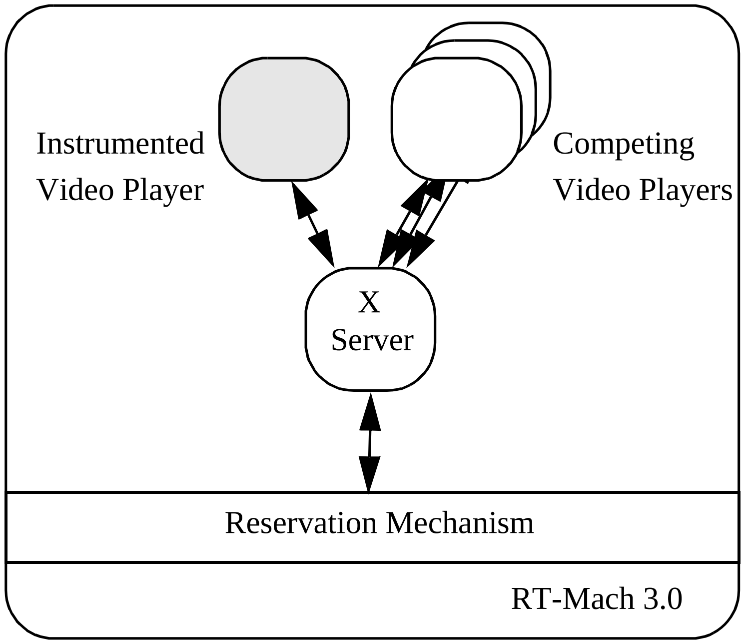

Figure 6-11: Software Configuration

Figure 6-11 shows the basic software structure that is used for all of the experiments in this section. There is an instrumented `QTPlay` application which may or may not have a reservation and which records timestamps for each frame at the beginning of the frame display computation and then again at the completion of frame display. Other instances of `QTPlay` may compete with this instrumented player. These are unreserved and continuously display frames as fast as possible (providing the maximum competition).

`QTPlay` can display frames at a particular period or in a continuous loop, and in all of the experiments below, the frame resolution is 160x120 pixels with 8 bits/pixel. The timing is specified by command-line arguments.

Experiment 9 is designed to illustrate the usage pattern for `QTPlay` with no competition for the processor. The parameters for `QTPlay` are given in Table 6-9. The period is 33 ms, which corresponds to a frame rate of 30 frames/second.

Program	#	Mode	Period	Reserved Computation	Reservation Period
`QTPlay`	1	Periodic	33 ms	0 ms	0 ms

Table 6-9: Experiment 9 Parameters

Experiment 10 is intended to show what can happen when `QTPlay` is executed under a time-sharing scheduler with a competing instance of the `QTPlay` program. The parameters for this experiment appear in Table 6-10. The `QTPlay` instance listed in the first row of the table is instrumented to provide timing information and the other just competes for the resources for displaying frames by continuously displaying frames as fast as possible.

Program	#	Mode	Period	Reserved Computation	Reservation Period
`QTPlay`	1	Periodic	33 ms	0 ms	0 ms
`QTPlay`	1	Continuous	N/A	0 ms	0 ms

Table 6-10: Experiment 10 Parameters

Experiment 11 is designed to show how well an instance of `QTPlay` with a reservation can coordinate with the reserved X Server to achieve a constant playback rate for frames. Table 6-11 gives the parameters for the experiment. The instrumented `QTPlay` application has a reservation and competition from one other unreserved `QTPlay` instance.

Program	#	Mode	Period	Reserved Computation	Reservation Period
`QTPlay`	1	Periodic	33 ms	14 ms	33 ms
`QTPlay`	1	Continuous	N/A	0 ms	0 ms

Table 6-11: Experiment 11 Parameters

Experiment 12 is similar to Experiment 10 in that it explores the behavior of an unreserved `QTPlay` with competition from 3 unreserved `QTPlay` instances rather than just one. For completeness, the parameters appear in Table 6-12.

Program	#	Mode	Period	Reserved Computation	Reservation Period
`QTPlay`	1	Periodic	33 ms	0 ms	0 ms
`QTPlay`	3	Continuous	N/A	0 ms	0 ms

Table 6-12: Experiment 12 Parameters

Experiment 13 is similar to Experiment 11; it looks at the behavior of a reserved `QTPlay` instance with three competing `QTPlay` instances. The parameters are given in Table 6-13. These last two experiments look at the behavior of unreserved and reserved `QTplay` applications under adverse conditions (intense competition from multiple unreserved X clients).

Program	#	Mode	Period	Reserved Computation	Reservation Period
`QTPlay`	1	Periodic	33 ms	14 ms	33 ms
`QTPlay`	3	Continuous	N/A	0 ms	0 ms

Table 6-13: Experiment 13 Parameters

6.2.3.2 Results

The results for Experiment 9 illustrate the timing behavior of a `QTPlay` application with no competition. Figure 6-12 shows these results. For each frame, the player records the starting time and ending time. The x-axis is the frame number, counting frames starting at the 200th frame through to the 400th. For each frame value, the difference between the start time and end time is computed, and the y-axis is this frame delay measured in milliseconds.

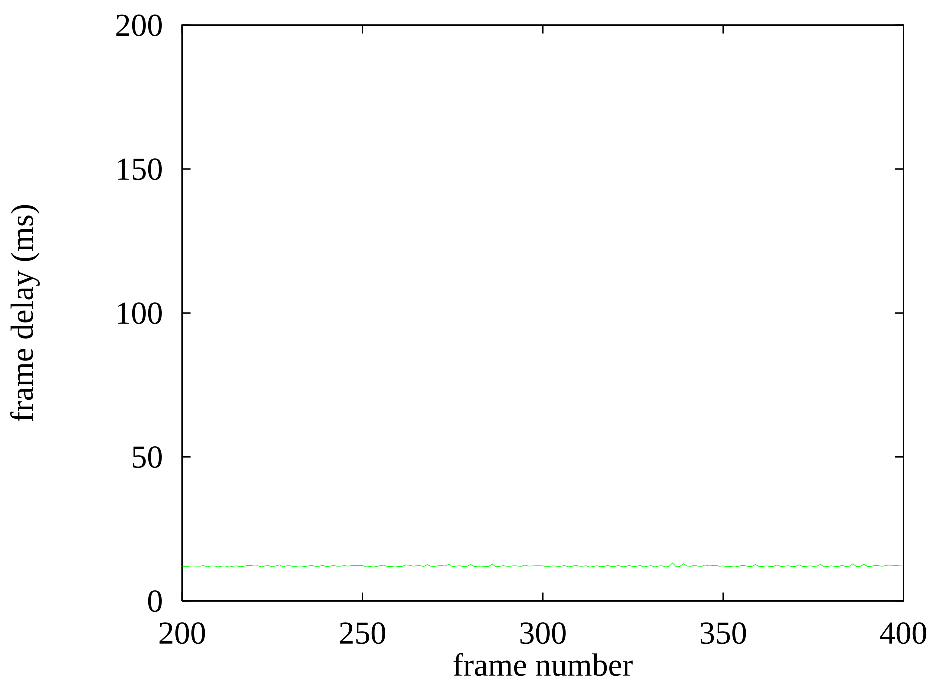

Figure 6-12: Experiment 9 Results

In the figure, the frame delay averages 12.1 ms with a 5-percentile of 11.8 and a 95-percentile of 12.6. This indicates that the software took an average of about 12 ms to perform all the computations necessary to display a frame when there was no competition for resources.

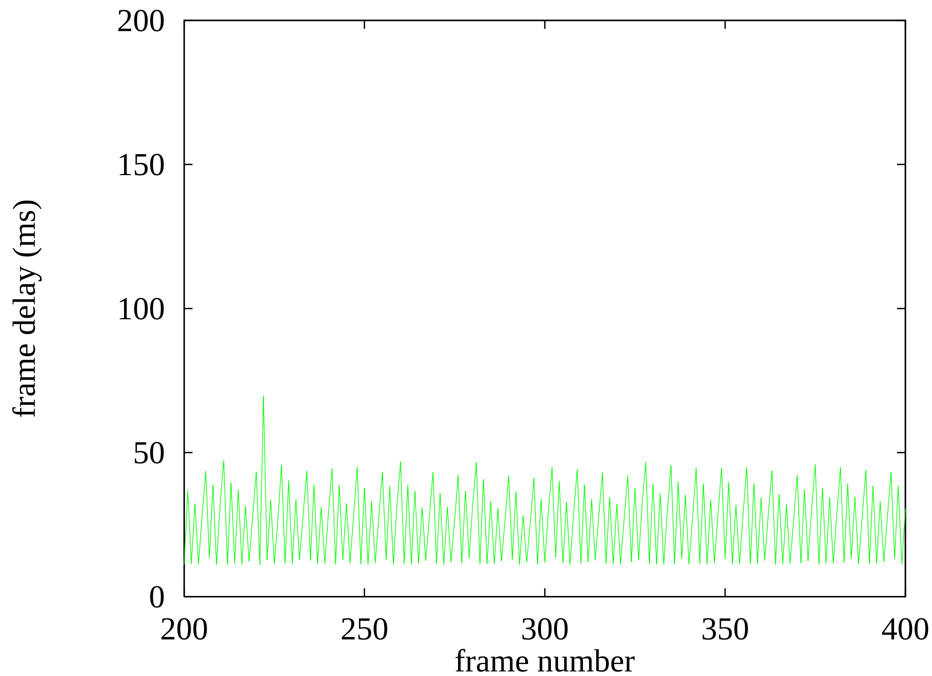

Figure 6-13: Experiment 10 Results

The results from Experiment 10 show a slightly different picture in Figure 6-13. Again, the x-axis is frame number, and the y-axis is frame delay measured in ms using the same method. With time-sharing scheduling and one competing `QTPlay`, the instrumented `QTPlay` sees quite a bit of interference in its frame delay time. The frame delay is much more variable. The average delay is 25.8 ms with a 5-percentile of 11.2 and a 95-percentile of 45.0.

In the results from Experiment 11, the instrumented `QTPlay` has a reservation, and its frame delay is much less variable even with competition from one unreserved `QTPlay` instance. Figure 6-14 shows the timing behavior. As before, the x-axis is frame number; the y-axis is frame delay in milliseconds.

The frame delay still has a bit of variation, but it is much less variable than the case where the `QTPlay` application is unreserved. The average delay is 19.4 ms with a 5-percentile of 14.7 and a 95-percentile of 24.3. So `QTPlay` is almost always able to display each frame within its 33 ms period.

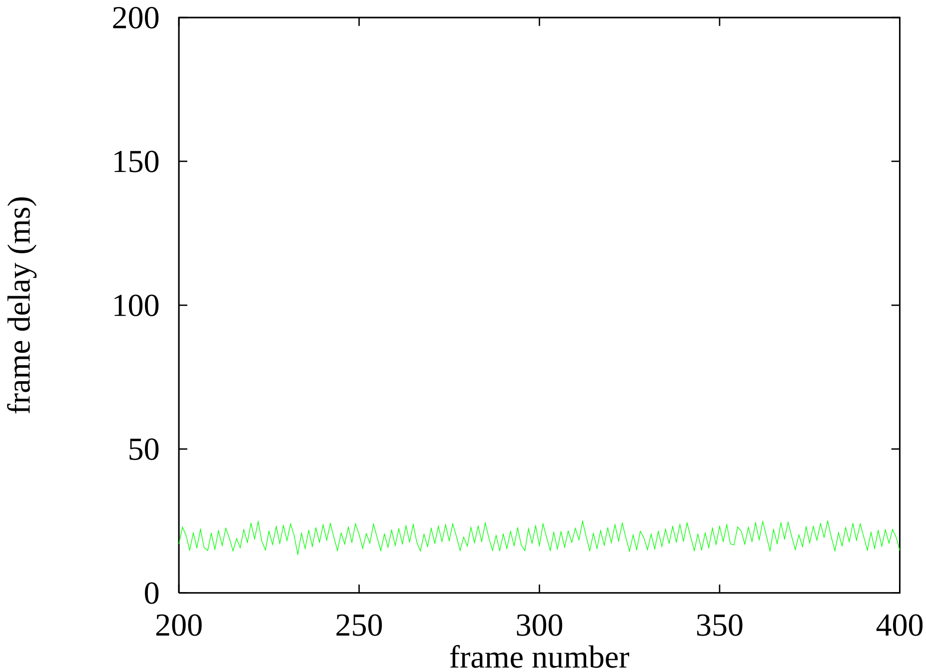

Figure 6-14: Experiment 11 Results

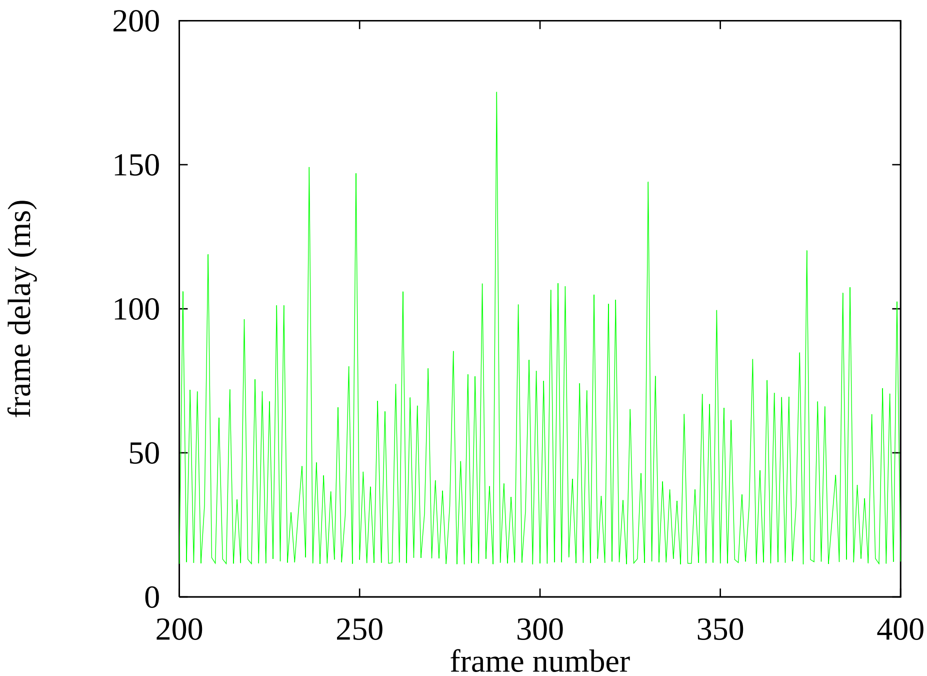

Figure 6-15: Experiment 12 Results

The results from Experiment 12 show how much the frame delay variation can be for an unreserved `QTPlay` instance that has three unreserved `QTPlay` applications competing to display frames. Figure 6-15 shows these results. As the figure shows, the variation in frame delay is quite large. The 5-percentile for the frame delay is 11.5 ms, and the 95-percentile is 107 ms with an average frame delay of 40.3 ms. A delay of 150 ms (which does not show up in the 95-percentile number but occurs a number of times during the test) or even 100 ms in a sequence of video frames is clearly noticeable to the human eye. Frame rates of 15 frame/second or more are required to sustain the illusion of smooth motion. This implies that with delays above 66 ms or so, the illusion of smooth motion may be destroyed.

In contrast, the results of Experiment 13, shown in Figure 6-16, demonstrate how well the reservation system can control the variability in frame delay for the reserved `QTPlay` application, even with much competition from unreserved instances of the `QTPlay` program. The frame delay in the figure is somewhat variable, but the variation is much less than in the case of the unreserved `QTPlay` with three competitors. The 5-percentile is 13.4 ms and the 95-percentile is 34.2 ms with an average of 20.7 ms. This is well below the target of the 60 ms period necessary for smooth-looking motion.

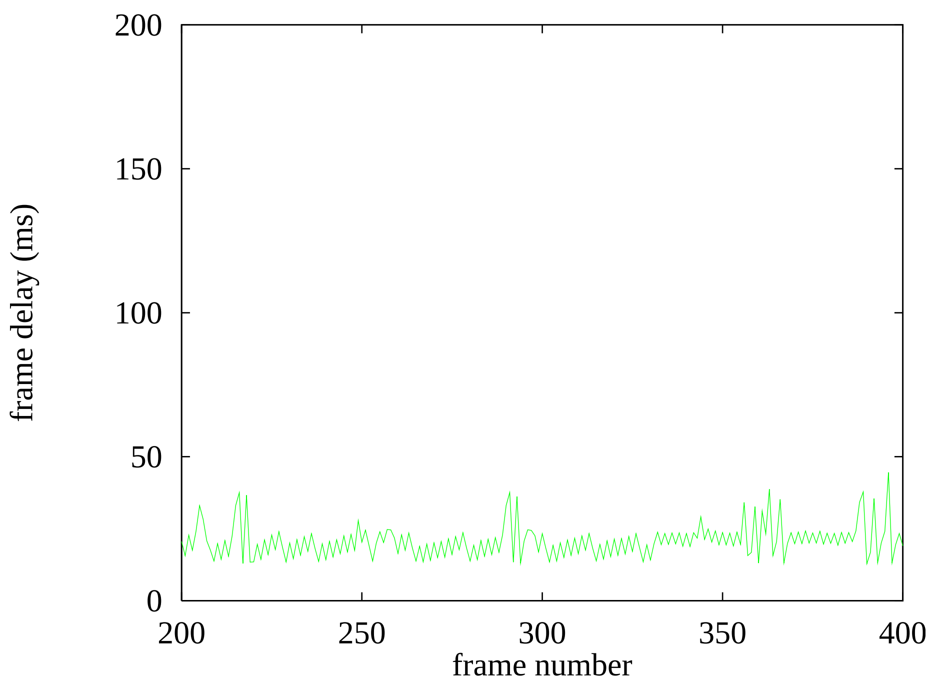

Figure 6-16: Experiment 13 Results

6.2.3.3 Analysis

The `QTPlay`/X Server experiments show that even with real applications like a QuickTime video player and the X server, the reservation system can provide predictable performance for real-time programs that must meet timing constraints to achieve satisfactory performance.

Experiment 9 shows the baseline behavior that is desired for the video player. This experiment has no competing activity, so there is no contention for resources and the behavior is very regular. Experiment 10 has a competing video player in addition to the instrumented video player, and with time-sharing scheduling, this competitor causes some scheduling delay in the instrumented player. In Experiment 12, there are three competitors, and the interference to the instrumented player is very bad. The player frequently has frame delays of 60 to 100 ms and even as high as 150 ms. These kinds of delays are clearly noticeable to a human observer of the video stream.

In Experiment 11, the instrumented video player is reserved with one competitor, and although there is a little fluctuation in the frame delay, it is limited to 24.3 for the 95-percentile. In Experiment 13, the reserved video player has competition from three unreserved video players. The frame delays show a little more variability, but are still limited to a 95-percentile of 34.2 ms compared to the 100 and even 150 ms delays experienced with time-sharing scheduling.

The question of why the reserved `QTPlay`/X Server combination suffered any delay arises. The reason is that the X Server was not implemented from scratch to use reserves. The extensions to allow it to use reserves did not completely restructure the request input queue, in particular. So the server reads requests from its input queue, orders them internally, and then performs the operations. If server's client interface code were completely rewritten to support reserves, the behavior would be comparable to that of the client/server synthetic benchmarks where the servers were designed from scratch.

6.2.4 mpeg_play/X Server

In addition to the QuickTime video player, a version of the `mpeg_play` decoder was modified to use reserves and coordinate with the reserved X Server. This decoder uses some simple usage measurement and adaptation techniques to tune the reservation parameters and timing parameters based on changing system conditions.

6.2.4.1 Methodology

The `mpeg_play` modifications were described in detail in the previous chapter, so the description here is brief. The player prefetches a video clip into memory to avoid interference in the file system. It requests a processor reservation and passes the reference to its reserve to the X Server. While it is executing, the decoder keeps track of its resource usage and timing characteristics, and it makes adjustments to the reservation parameters and/or period of the program based on usage.

Experiment 14 is designed to determine whether the `mpeg_play` decoder can successfully modify its reservation parameters and/or behavior based on existing conditions. The decoder starts executing with a reservation that is too small for its computation time. Competition is then introduced in the form of reserved and unreserved activities.

6.2.4.2 Results

The behavior of the mpeg_play application under the conditions of Experiment 14 is illustrated in Figure 6-17. The decoder is able to tune its reservation parameters based on run-time information and stabilize its own behavior.

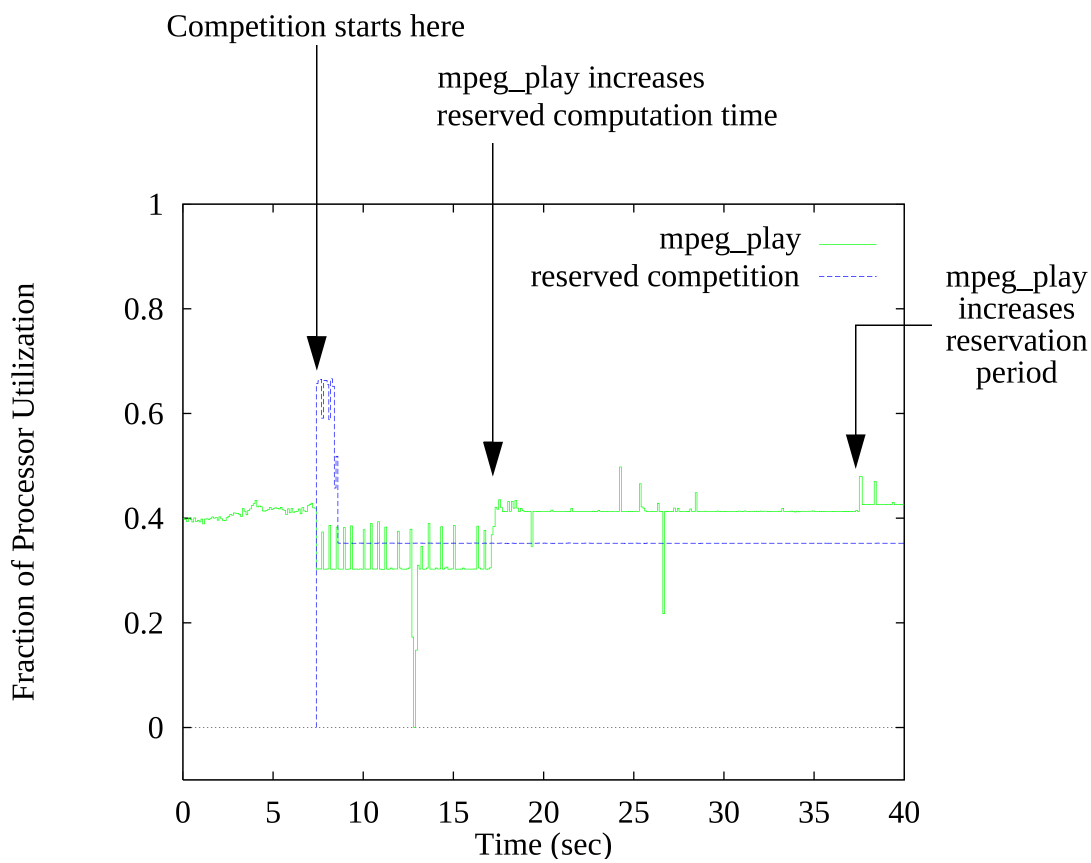

Figure 6-17: Experiment 14 Results

Figure 6-17 shows the processor utilization of the mpeg_play decoder over a period of 40 seconds. The x-axis is time in seconds, and the y-axis is processor utilization. During the first 7 seconds, mpeg_play averages about 40% of the processor, even though its reservation is only 30 ms every 100 ms. Since there is no competition, it consumes 40%. At about 7 seconds into the experiment, a reserved program (shown in the graph) and several unreserved programs (not shown) are introduced. The usage of mpeg_play immediately drops to its reserved level of 30% of the processor. This is not enough to sustain its previous frame rate, so some frames are dropped.

The reserved activity, which had a usage spike at the time it started, settles down to a constant 37%. The spike occurs since the time-sharing algorithm initially allows the new program to get more cycles than its reservation. After consuming a large percentage of the processor, however, the reserved activity no longer gets additional cycles from the time-sharing algorithm, and the usage flattens.

After several seconds, `mpeg_play` realizes its frame rate has fallen and attempts to increase its reservation. At about 17 seconds into the experiment, the decoder increases its reservation to 41 ms reserved every 100 ms, and its frame rate increases accordingly. Again at about 37 seconds into the experiment, the decoder changes its reserved computation time to 47 ms and its reservation period to 111 ms to fine-tune the reservation even further.

6.2.4.3 Analysis

This experiment demonstrates that an application can adjust its reservation parameters and adapt its behavior based on the usage information from the reservation mechanism. In this case, the initial reservation of the `mpeg_play` application did not have to be accurate since the application automatically adjusted the reservation levels based on usage measurements.

6.2.5 Protocol processing workloads

The experiments in this section explore the real-time behavior of the socket library (called `libsockets`) protocol processing. The behavior obtained by an application using the socket library is compared to the behavior using the UX Server's socket service under both time-sharing scheduling and reserves.

6.2.5.1 Methodology

The `libsockets` experiments use the same software environment as the experiments with independent tasks and client/server workloads. There is a configuration manager that reads the task set specification and creates the specified programs. In addition to the workload programs introduced so far, these experiments use the following programs:

- `stdio` - Creates a periodic thread that calls a sequence of file operations.
- `udps` - Creates a periodic thread that sends some number of packets (specified in the program computation field) in each period. This program uses the UX Server to send packets.
- `udpls` - Creates a periodic thread that sends some number of packets in each period. This program uses libsockets to send the packets rather than interacting with the UX Server.
- `udpr` - Creates a periodic thread that receives some number of packets in each period using the UX Server to receive the packets.
- `udplr` - Creates a periodic thread that receives some number of packets in each period using libsockets.

Experiment 15 is designed to show how a packet-sending activity can be disturbed by competition from a combination of compute-intensive and I/O-intensive activities, especially when all of those activities use system services which interact with each other as they do in the UX Server. The task set for this experiment is given in Table 6-14. It shows two udp senders (`udps`) with slightly different workloads. The competition for this experiment (as for the next three experiments) consists of five compute-intensive `arith` programs and five I/O-intensive `stdio` programs. In this experiment, the packets sent by the `udps` programs are received on a remote machine by yet another program that records a timestamp when the each packet arrives. This program buffers the timestamps and dumps them out at the end. The timestamp data can be used to judge whether the packet senders were able to send their packets out as desired or not.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
`udps` (A)	1	4 pkt	40 ms	0 ms	40 ms
`udps` (B)	1	2 pkt	40 ms	0 ms	40 ms
`arith`	5	various	various	0 ms	40 ms
`stdio`	5	various	various	0 ms	40 ms
`monitor`	1	N/A	20 ms	1 ms	20 ms

Table 6-14: Experiment 15 Parameters

Experiment 16 is designed to determine whether the packet-sending applications can send their packets on time when using `libsockets` for their network protocol processing. The parameters are given in Table 6-15. This experiment differs from Experiment 15 in that the UDP packet senders use libsockets instead of the UX Server and they have reservations instead of being scheduled by the time-sharing scheduler. The competition is the same: five compute-intensive programs and five I/O-intensive programs.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
`udpls` (A)	1	4 pkt	40 ms	10 ms	40 ms
`udpls` (B)	1	2 pkt	40 ms	6 ms	40 ms
`arith`	5	various	various	0 ms	40 ms
`stdio`	5	various	various	0 ms	40 ms
`monitor`	1	N/A	20 ms	1 ms	20 ms

Table 6-15: Experiment 16 Parameters

The purpose Experiment 17 is to determine whether a UDP packet-receiving program that attempts to receive a number of packets periodically can meet that objective. The `udpr` program attempts to receive some number of packets in each period. This program receives packets through the UX Server and runs without a reservation. The competing activities are identical to the previous two experiments. Table 6-16 presents the parameters for this experiment.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
`udpr` (A)	1	4 pkt	40 ms	0 ms	40 ms
`udpr` (B)	1	3 pkt	40 ms	0 ms	40 ms
`arith`	5	various	various	0 ms	40 ms
`stdio`	5	various	various	0 ms	40 ms
`monitor`	1	N/A	20 ms	1 ms	20 ms

Table 6-16: Experiment 17 Parameters

Experiment 18 is designed to determine whether a reserved packet receiver that uses libsockets can predictably execute periodically to receive a number of packets. The `udplr` program is a periodic packet receiver that uses `libsockets` instead of the UX server. This experiment includes the same competition from compute-intensive and I/O intensive tasks as the other experiments in this section. The parameters are given in Table 6-17.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
`udplr` (A)	1	4 pkt	40 ms	0 ms	40 ms
`udplr` (B)	1	3 pkt	40 ms	0 ms	40 ms
`arith`	5	various	various	0 ms	40 ms
`stdio`	5	various	various	0 ms	40 ms
`monitor`	1	N/A	20 ms	1 ms	20 ms

Table 6-17: Experiment 18 Parameters

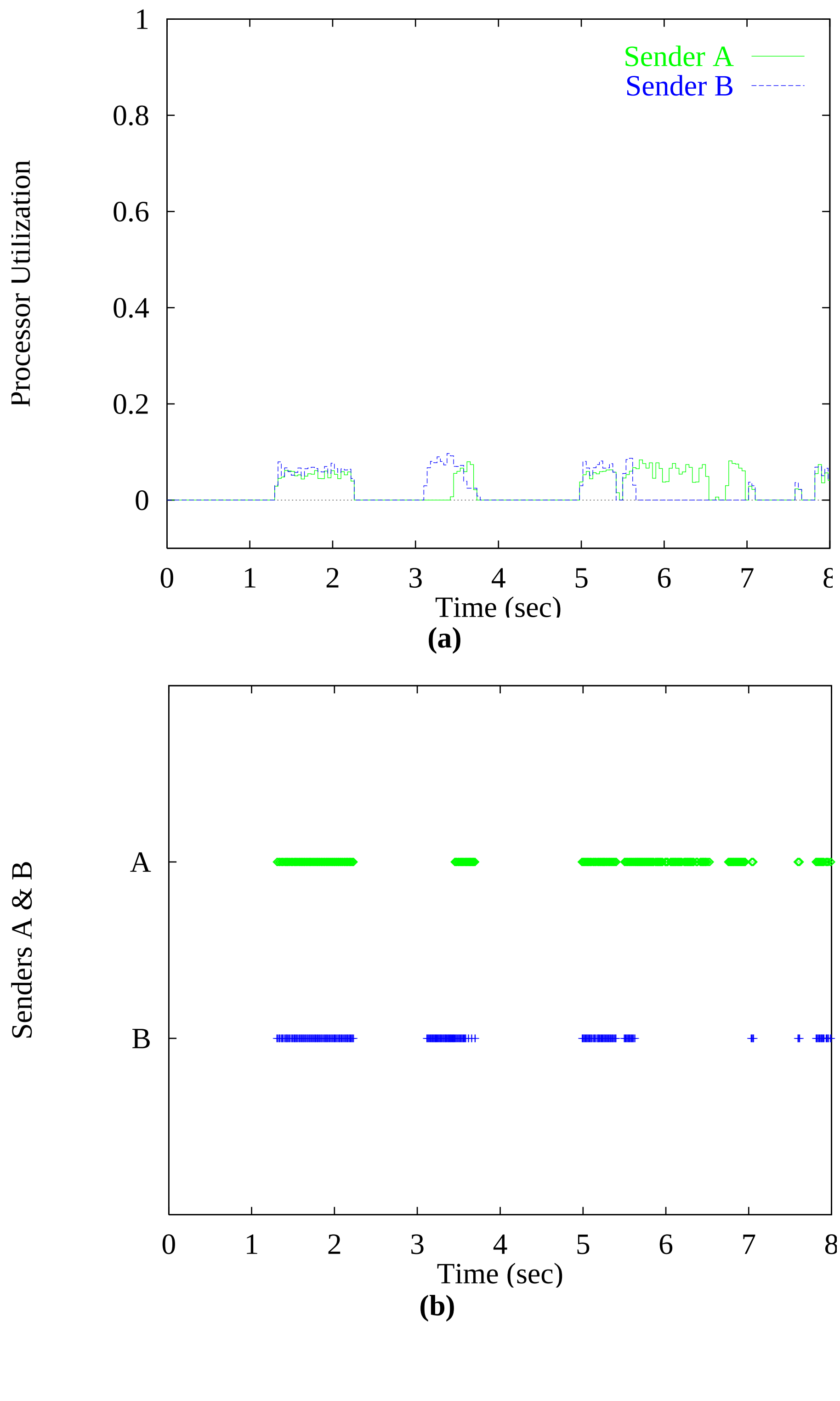

Figure 6-18: Experiment 15 Results

6.2.5.2 Results

The results of Experiment 15 are shown in Figure 6-18. Part (a) of the figure shows the processor utilization over time for the two packet senders, which use the UX Server implementation of sockets and run without reservations. The x-axis is time in seconds, and the y-axis is processor utilization. The two programs, denoted “Sender A” and “Sender B”, show a very erratic usage pattern. Frequently, the usage drops to zero for over 1 second. The average utilization for Sender A is 0.023 with a 5-percentile of 0 and a 95-percentile of 0.0741. The average utilization for Sender B is 0.0193 with a 5-percentile of 0 and a 95-percentile of 0.0796. The dispersion is clearly substantial for these applications, and that dispersion in utilization achieved translates directly into missed deadlines.

Figure 6-18(b) shows the record of timestamps that were received by a remote receiver for both senders. The x-axis in the graph is time in seconds, and there are two horizontal lines, one for Sender A and the other for Sender B. A mark on the line corresponding to Sender A at a particular time indicates that a packet arrived at that time and likewise for Sender B. This graph shows that the packets were received on the remote host sporadically. The pattern of packet receptions corresponds closely with the pattern of usage seen in part (a) of the figure. The largest gaps between received packets were 1.28 seconds and 1.22 seconds for Sender A and 1.93 seconds and 1.40 seconds for Sender B. These senders are attempting to send packets every 40 ms, so clearly they are not able to schedule the message sending activity as desired.

The results of Experiment 16 appear in Figure 6-19. In this case, the senders have reservations and use the `libsockets` library to avoid depending on the UX Server for networking. The graph shown in part (a) of the figure shows time in seconds on the x-axis and processor utilization on the y-axis. The processor utilization for both of the senders is very regular, indicating that in each reservation period, the programs were able to send the packets they were supposed to send. There are no long intervals of zero usage as in the previous case. The average utilization of Sender A is 0.103 with a 5-percentile of 0.101 and a 95-percentile of 0.110 indicating a very tight distribution around the average. Likewise for Sender B, the average utilization is 0.0591 and the 5-percentile is 0.0582 with a 95-percentile of 0.0624. This is also a tight distribution.

The graph in Figure 6-19(b) supports the conclusion that the senders in this experiment are able to send their packets very regularly in each period. As before, the x-axis is time in seconds and two horizontal lines indicate the timestamps for packets received from the two senders. A point on the line corresponding to Sender A represents a packet that was received at the associated time. In this experiment, packets are received very regularly from each sender. There are no significant gaps in the reception pattern. The maximum gaps for packets received from Sender A are 0.0519 seconds and 0.0423 seconds, and the maximum gaps for packets received from Sender B are 0.0438 seconds and 0.0435 seconds. So the conclusion is that the combination of libsockets with reserved resources yields predictable behavior for packet senders.

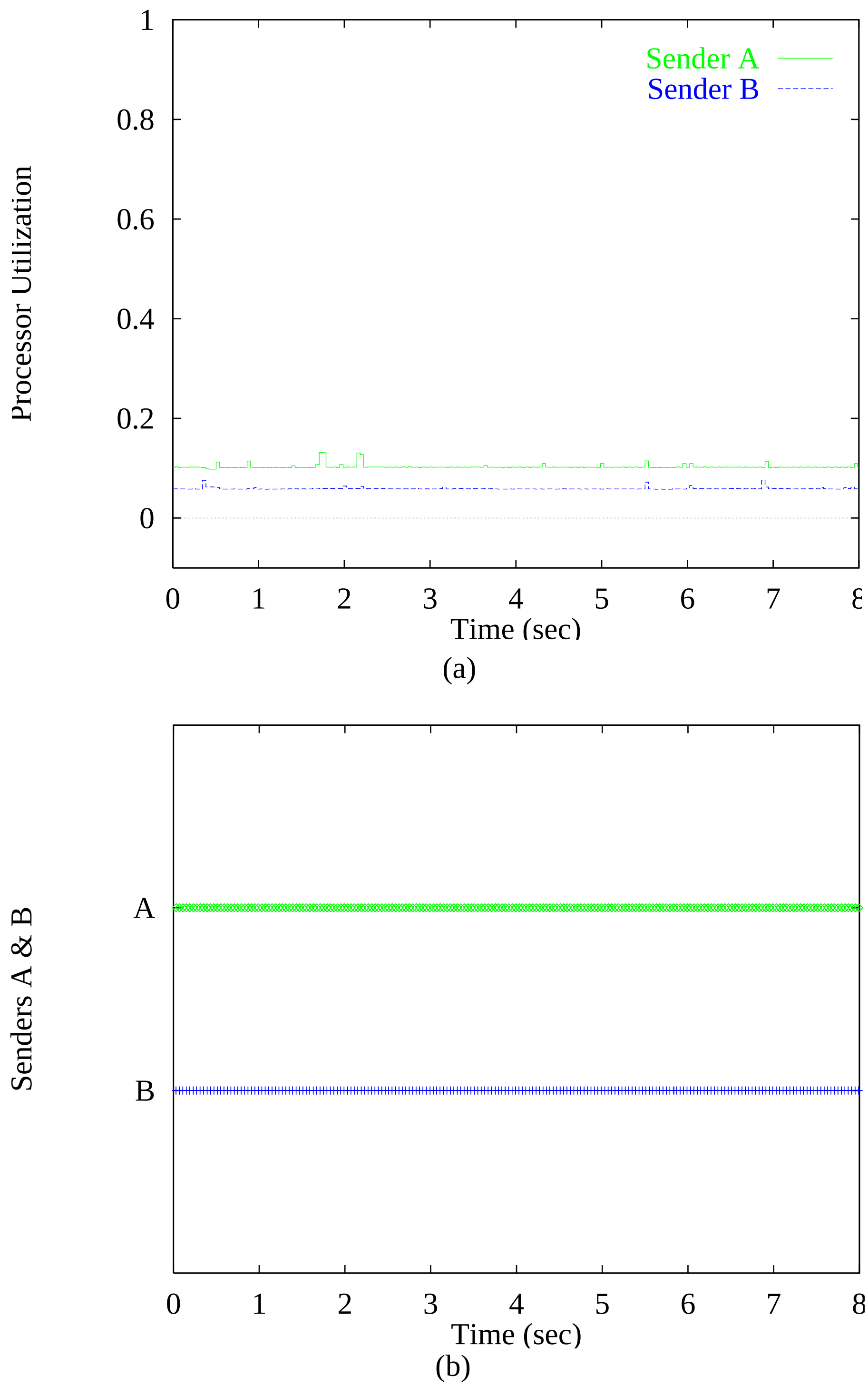

Figure 6-19: Experiment 16 Results

Figure 6-20 shows the results of Experiment 17. In this case, a remote sender periodically sends packets to the two receivers described in the task set. In this experiment, the two receivers are unreserved and use the UX Server's implementation of sockets. The graph has time in seconds on the x-axis and processor utilization on the y-axis. The behavior is very erratic. The remote host sends packets periodically, but the receiver is not always able to run long enough to receive the packets. Receiver A has an average utilization of 0.0134 with a 5-percentile of 0 and a 95-percentile of 0.0617. This is not the kind of timely behavior desired in the packet receiver. Receiver B has an average utilization of 0.0220 with a 5-percentile of 0 and a 95-percentile of 0.0726. Again, the dispersion is significant. The usage for both receivers frequently drops to zero, indicating that the packets are being dropped for significant periods of time.

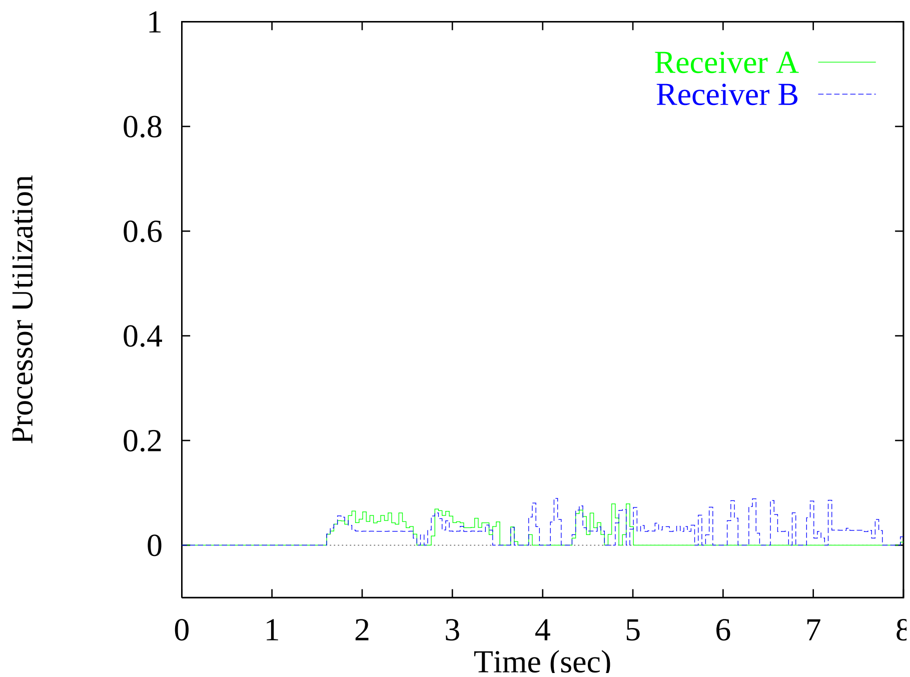

Figure 6-20: Experiment 17 Results

The results of Experiment 18 are shown in Figure 6-21. Again, a remote sender periodically sends packets to two receivers which have reservations and which use the `libsockets` implementation of sockets. The graph has time in seconds on the x-axis and processor utilization on the y-axis. The usage functions of the two receivers are plotted over the duration of the experiment. The average utilization for Receiver A in this case is 0.149 with a 5-percentile of 0.139 and a 95-percentile of 0.173. This indicates a fairly tight distribution. Receiver B has an average utilization of 0.130 with a 5-percentile of 0.115 and a 95-percentile of 0.140. Again the utilization achieved is quite consistent across periods for the duration of the test. It is clear that the receivers do not drop to very low utilizations for significant intervals of time. Their relatively constant usage indicates that they are able to process the incoming packets in a predictable manner.

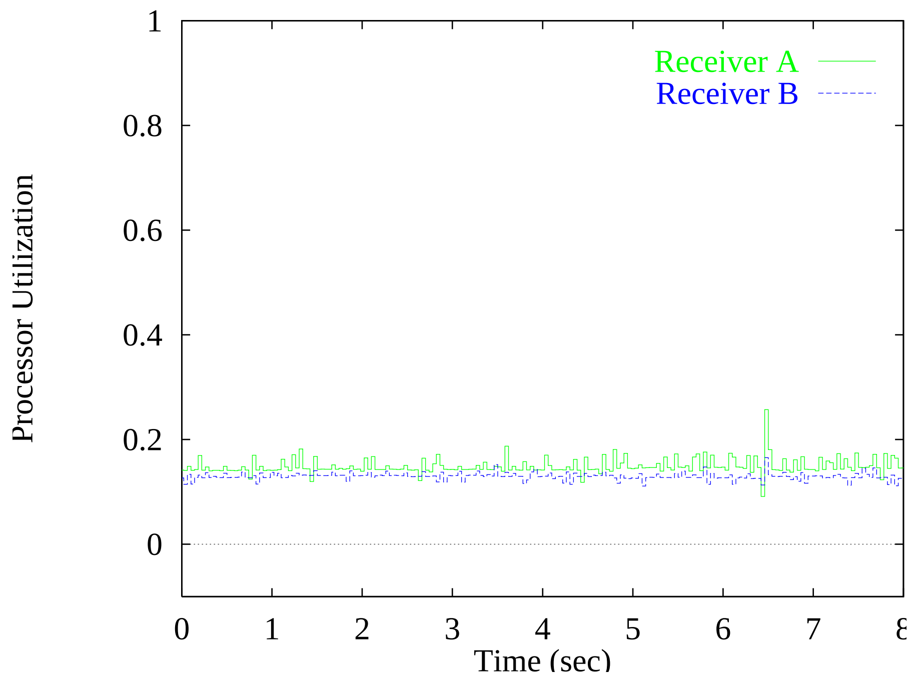

Figure 6-21: Experiment 18 Results

6.2.5.3 Analysis

The results from the `libsockets` experiments showed that when packet senders and receivers ran in time-sharing mode using the socket implementation provided by the UX Server, their behavior was erratic when other programs were competing for the processor and access to other services provided by the UX Server. Reserved packet senders and receivers that used `libsockets` for handling network packets had much better behavior. They were able to execute periodically and perform each sending or receiving computation by the end of the corresponding period. Other experiments (not presented here) showed that the behavior of reserved programs that used UX sockets was just as bad as that of unreserved programs with UX sockets. Also, unreserved programs that used `libsockets` exhibited unpredictable behavior when executing with competition as well. These experiments indicate that using a reservation mechanism or a `libsockets` mechanism alone does not ensure predictability in programs; both mechanisms are needed.

6.3 Scheduling cost

This section addresses the scheduling costs of predictable programs that can meet their timing constraints under the reservation system. It looks at measured aggregate costs for the system as well as measurements of scheduling operations that contribute to the costs. Such measurements enable cost projections to be made for specific task sets.

6.3.1 Measured aggregate scheduling cost

The reservation system ensures that reserved resources will be available to enable real-time programs to meet their timing constraints, but this predictability has costs associated with it. In particular, the accurate measurements and the timers necessary for enforcement take some time. In measuring the aggregate scheduling cost, the intention is to determine how much time is consumed by scheduling costs in the case of a reserved periodic thread compared to that of an unreserved thread.

6.3.1.1 Methodology

Experiment 19 is designed to measure the scheduling cost for a periodic thread as the period varies. The task set includes the periodic thread and an “idle” thread that runs in the background to consume all processor time not consumed by the periodic thread and the system’s housekeeping activities. The system scheduling cost is taken to be the total time of the test minus the time consumed by the periodic thread and the idle thread. This scheduling cost includes context switch times, associated cache effects, as well as the cost of timers and clock operations for the reservation mechanism.

One series of tests measures the scheduling cost for a reserved periodic thread whose reservation period ranges from 20 ms to 200 ms. Most of the cost for the reservation system is a fixed cost in each period, so a longer period implies a relatively smaller cost. The other series of tests measures scheduling cost for an unreserved periodic thread with a period that varies from 20 ms to 200 ms.

6.3.1.2 Results

The results of the scheduling cost measurements are presented in Figure 6-22. The graph consists of two functions: the scheduling cost associated with a reserved periodic thread as a function of period and the scheduling cost associated with an unreserved periodic thread as a function of period. The x-axis is the period in milliseconds, and the y-axis is the percentage of the processor that is lost to scheduling costs.

The scheduling cost for the reserved thread starts out about 3% for a reservation period of 20 ms and drops off as the reservation period is increased. For a 100 ms reservation period, the scheduling cost is about 0.5% and for a 200 ms reservation period, it is about 0.2% For the unreserved thread, the scheduling cost is smaller, starting at about 2.2% for a 20 ms period. The cost drops off to about 0.5% for a 100 ms period and then to about 0.1% for a 200 ms period.

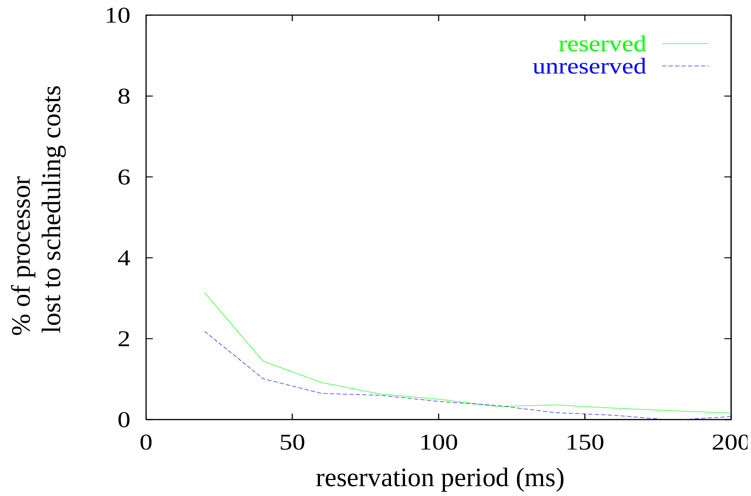

Figure 6-22: Scheduling Cost

6.3.1.3 Analysis

The scheduling cost measurements indicate that the cost of threads with very small reservation periods (smaller than about 30 ms) grows somewhat as the reservation period decreases. For reservation periods in the range of 40 to 100 ms, which would be an appropriate range for many audio and video applications for example, the scheduling cost is acceptable.

The scheduling cost is relatively high for the reservation system because the clock/timer card used in the experiments is very sensitive to the timing of loads and stores in its control and data registers. The card is used quite often in the reservation system to read a free-running clock, set an interrupting timer, or cancel a timer. Since each of these operations requirements multiple reads and stores to device registers and since the device driver for the card contains many delay loops required to synchronize properly with the card, much time is wasted. With clock and timer support from a better card, the scheduling cost should be significantly lower.

6.3.2 Individual operations

This section presents measurements of the individual internal operations used by the reservation mechanism. These measurements can be used to project scheduling costs for task sets.

6.3.2.1 Reserve Switch

During a context switch, the system must switch the reserve to which it is charging computation time as it switches the thread that is running on the processor. This involves updating usage accumulators in the old reserve and possibly setting the overrun timer for the next reserve. The four measured cases for the reserve switch are:

- Neither the old or new activity was reserved - Just update the usage accumulators.
- The old activity was reserved - Cancel the overrun timer for the old activity and update usage.
- The new activity is reserved - Set up and arm the overrun timer for the new activity and update usage
- Both activities are reserved - Cancel the old overrun timer, set up the new overrun timer, and update usage.

The following table gives the measurements for these cases.

Action	Duration
Neither old nor new activity reserved	23 µs
Old activity reserved	100 µs
New activity reserved	100 µs
Both activities reserved	180 µs

Table 6-18: Reserve Switch

6.3.2.2 Overrun and Replenishment Timers

The measured cost for handling an overrun timer includes identifying the timer, setting some state in the reserve, and initiating a context switch. It does not include the cost of the context switch itself.

The cost for handling a replenishment timer includes identifying the timer, setting some state in the reserve, resetting the timer for the next reservation period boundary, and possibly resetting the overrun timer. Four different cases for replenishment timer handling were measured:

- A reserve with a reservation whose computation allocation had been depleted during the period (the overrun timer had expired for this activity), and it was waiting for a new allocation.
- A reserve with a reservation whose computation allocation had not been depleted.
- A reserve with no reservation whose replenishment timer expired while it was running.
- A reserve with no reservation which was not running at the time the replenishment timer expired.

The following table shows the measurements for the overrun timer and each of the replenishment timer cases.

Action	Duration
Handle overrun timer	130 µs
Reserved and waiting for new allocation	170 µs
Reserved but not waiting	140 µs
Unreserved and running when timer expired	140 µs
Unreserved and not running when timer expired	140 µs

Table 6-19: Replenishment Timer

6.3.2.3 Usage checkpoints

This section gives measurements of the system primitives that extract usage information from the kernel. The reserve usage data from the reservation system implemented in RT-Mach come in two forms: the current accumulated usage of a reserve, and the accumulated usage as of the last reservation period boundary. The record of the accumulated usage of a reserve taken at a reservation period boundary is called a checkpoint. The reservation system offers two system primitives for retrieving usage data: the first gives the data for a single checkpoint along with the current accumulated usage, and the second give the last 20 checkpoints from the last 20 reservation period boundaries. The following table gives the measured costs for both of these system primitives.

Action	Duration
Retrieve single checkpoint	130 µs
Retrieve 20 recent checkpoints	220 µs

Table 6-20: Checkpoint Cost

6.3.2.4 Analysis

The measurements described in this section can be used to estimate the cost associated with running specific task sets on the reservation system. The replenishment timer costs occur in every period of every reserved activity, and these numbers can help project the impact of having many tasks with small reservation period. The overrun timer costs detail the penalty associated with a program whose computation does not closely match its reservation. The checkpoint costs can be used in estimating the overhead for a monitor and in choosing timing properties of a monitor to balance accuracy of information with overhead cost.

6.4 Chapter summary

This chapter addressed the questions of whether the reservation system supports predictable application behavior and how much the predictability costs in terms of scheduling overhead. To show that the reservation system supports predictable behavior for applications, several experiments were done using task sets consisting of independent compute-bound synthetic workloads as well as client/server task sets with synthetic workloads. In an experiment with three reserved programs and five unreserved competitors, the reserved programs achieved measured processor utilizations that had 5-percentiles and 95-percentiles within 3-7% of their average utilizations, indicating that they were able to get their processor reservations with very little variance in their computation times. In the client/server experiments, even a case where an unreserved client was competing with two reserved clients for the same server (along with other independent unreserved programs competing for the processor), the reserved clients were able to achieve their timing constraints. The 5-percentiles and 95-percentiles for the two reserved clients in this case where within 5% of their average utilizations. These experiments showed that the reservation system guarantees very tight distributions of processor utilization even with different types of computation and with different combinations of client/server interactions.

Additional experiments used a reserved QuickTime video player and a modified version of the X Server to show that reserved applications can coordinate with shared servers to satisfy timing constraints on computations such as displaying video frames. In these experiments, the reserved players (even with interference from competing non-real-time X clients) had processor utilization measurements with 5-percentiles and 95-percentiles within 65% of their average utilizations. The utilization distributions were not as tight as the synthetic client/servers because the internal structure of the X Server is not ideally suited to reserve propagation and charging to clients' reserves. However, the performance of the reserved players was still much improved over that of unreserved players which had measurements with 5-percentiles and 95-percentiles that were as much as 166% of their average utilizations.

Experiments with libsockets showed that with an appropriate protocol processing structure, packet senders and receivers could achieve predictable behavior. The reserved senders in the experiments had processor utilization measurements with 5-percentiles and 95-percentiles that were within 7% of the average utilizations, and the reserved receivers had 5- and 95-percentiles that were within 16% of the average utilizations. This is compared with unreserved senders and receivers that had 5- and 95-percentiles of up to 360% of their average utilizations.

The scheduling costs of the reservation system were measured by running a periodic thread and measuring the idle time left over to find the scheduling cost. The results from several experiments with reserved and unreserved periodic threads with different periods show that the scheduling cost of reserved threads is typically about twice that of unreserved threads.