Performance Measurements of the Multimedia Testbed on Real-Time Mach

Roger B. Dannenberg, David B. Anderson,
Tom Neuendorffer, Dean Rubine

April 1994
CMU-CS-94-141

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Multimedia has generated widespread interest in real-time support within general purpose operating systems. Multimedia also places new demands on operating systems for interprocess communication. The Multimedia Testbed is a set of applications that stress consistent low-latency response and efficient interprocess communication for large blocks of data. The Multimedia Testbed was ported to Real-Time Mach in the hopes of providing predictable low-latency response and, consequently, good synchronization and low jitter as required for multimedia applications. Our work compares the performance of Real-Time Mach with that of Mach 3.0. Although the fixed-priority scheduling of Real-Time Mach is a substantial improvement, user threads are still preempted by device drivers, and the overall real-time performance is not suitable for multimedia applications. We discuss areas where Real-Time Mach needs improvement.

This research was performed by the Carnegie Mellon Information Technology Center and supported by the IBM Corporation.

Keywords: Performance Measurements, Multimedia, Real Time, Operating Systems, Interprocess Communication

1. Introduction

Multimedia support is necessary for increasing numbers of applications. The use of video for communication and documentation, the use of voice, music and sound effects in user interfaces, and the general trend towards greater bandwidth graphics and images places new demands upon systems. The ‘‘fair’’ scheduling and timesharing paradigms are no longer appropriate for multimedia personal computers.

One way to achieve real-time support for multimedia is to use a specialized real-time operating system. This approach has the drawback that existing applications would have to be ported. A more promising direction is to emulate existing operating systems using a microkernel and OS server approach. This approach supports existing applications and can even allow multiple operating system interfaces to coexist on a single machine. Our work assumes this microkernel approach, so we run all of our experiments with a Unix Server process providing a ‘‘normal’’ Unix environment.

In a previous report [Dannenberg 93a] we described an evaluation of the Mach Microkernel [Accetta 86] as a base for multimedia support. For this evaluation, we ported multimedia software from other systems and measured their performance. Rather than just measure the timing of individual operations, we have focused on ‘‘end-to-end’’ performance measurements in a system configured for normal use. We also measure individual operations as much as possible to explain the overall performance we observe.

This is often much more difficult than it seems, because unexpected interactions often occur. For example, when we measure the time it takes to process a clock interrupt on an unloaded machine, the clock interrupt handler and data are always in the cache. When we load the machine with a process that writes to many addresses (thus invalidating all the cache lines), the clock interrupt time increases by about a factor of three. This illustrates that interactions can have a dramatic effect upon measurements.

In general, we are concerned with two types of measurement. The first is latency, in other words, how quickly the system can respond to IO events and timer events. We have used music programs that receive and send MIDI events as a vehicle for latency measurements. Interactive music systems require latencies in the range of a few milliseconds, so music is probably the most critical multimedia application with respect to latency. The other type of measurement is interprocess communication. For these measurements, we implemented digital video applications that move data from the file system through a control application, a synchronization server, and finally to a window manager.

Using Mach 3.0, we used fixed-priority threads, a high resolution timer, a ‘‘user-level’’ device driver, and shared memory interfaces to improve the real-time properties of our programs. We used these same mechanisms for this study, except instead of fixed-priority threads, we used RT Threads. RT Threads add a new set of fixed-priority classes to the existing classes of Mach 3.0. From our perspective, the main advantage of RT Threads is that they run at a higher priority than the Unix Server or any other system threads.

Most of our timings are performed on a 33Mhz i486 processor running a Mach 3.0 MK69 Microkernel with a Mach 3.0 UX36 Unix server on a Gateway 2000 486/33C computer. The video measurements of Sections 6 and 7 were performed with a 66MHz i486-DX2 processor running on a Gateway 2000 4DX2-66V. In most cases, tests were done with machines running network servers and the Andrew file system, both of which impose significant loads that are difficult to predict.

For this study, we used our previous applications and some new ones:

1. A MIDI note generator that generates equally-spaced MIDI messages. By timestamping the MIDI data on another machine, we can measure jitter in the MIDI stream caused by timer-to-thread latency in RT Mach.

2. An audio synthesizer. In this application, incoming MIDI data triggers the software synthesis of digital audio by a user thread. In this case, the thread must be scheduled periodically with low latency and many messages containing digital audio packets must be delivered to an Audio server.

3. Tactus/Digital Video. The Tactus system consists of (1) a graphical user interface toolkit extended with support for concurrent computation and timestamping of media, and (2) a synchronization server which presents timestamped media to devices. We hoped that RT Mach's priority-based scheduling and priority inheritance would lead to more accurate timing in Tactus.

1.1. What Should We Expect?

In the previous work based on Mach 3.0, there was little reason to expect good real-time performance. The Unix server has very unpredictable performance, so we should not expect real-time performance from any of its clients. A Mach fixed-priority thread will not preempt an ordinary thread until the end of its quantum, so there is a built-in latency. There are also fixed-priority threads in the Unix server that interfere with applications. Finally, there are kernel threads such as the `io_done` thread that do not have fixed priority and can become very low in priority. The kernel itself is non-preemptive, and there are some long paths that take an indeterminate amount of time to execute.

A common model for real-time systems is that there are a number of threads and a scheduler that determines when threads will run. The behavior of the scheduler is the main factor that determines real-time performance. Other factors, such as interrupt handlers that preempt threads and interaction between threads are considered to be of less importance. Under this model, Mach 3.0 is flawed in that the Unix Server has long-running, high-priority threads that will preempt real-time application threads, and this results in erratic real-time performance.

One possible solution to the Mach 3.0 problem would be to rewrite the Unix Server to avoid the improper use of the highest priority levels. RT Mach overcomes this problem in a different way by offering a new class of threads (RT Threads) and a corresponding new class of priorities. Although the old Mach 3.0 priority levels are retained, the new RT Thread priorities are higher. When the Unix Server makes a call to set a thread to the highest priority, the RT Mach kernel gives it the highest Mach thread priority, which is strictly lower in priority than the RT Mach threads. Thus, an application written with RT Mach threads can run at the highest priority of any thread in the system. According to the model that scheduling is the most important determinant of real-time performance, RT Threads should allow the proper assignment of priorities and result in good real-time performance.

If this model is correct, one might hope for better performance with RT Mach. As it turns out, the model is only partially correct. In a sense, our work is a study of the limitations of this model. Although RT Mach addresses the thread priority and scheduling problems of Mach 3.0, there are other performance factors not addressed by RT Mach. These factors are important, so the overall performance is still not adequate for the sorts of multimedia applications we studied. It is important to keep in mind that real-time systems are generally measured in terms of worst-case performance. If there are five different ways in which the system can block a thread for 100ms, and you reduce four of them to 1ms, then users will see *no net improvement*. The worst case is still 100ms.

We note that many multimedia programs do pause, glitch, flicker, jump, and pop due to failures to compute or transfer data in a timely fashion. To the extent that this erratic, unpredictable behavior is acceptable, our measurements will be less meaningful. In many cases we give averages or histograms to give some idea of the disparity between the expected and worst cases.

RT Mach has addressed some, but not all of the problems we encountered with Mach 3.0. The problems which have been solved include:

- Thread Priority. RT Threads are essentially a new class of fixed-priority threads that run before the ordinary Unix server threads are scheduled.

- Synchronization. The C Threads package in Mach 3.0 uses spin-locks to avoid expensive kernel calls. This optimizes the expected time to execute a synchronization primitive, but leads to poor worst-case performance. Furthermore, if a high-priority thread spins on a lock, the lock holder may never get a chance to release the lock, leading to deadlock. Thus, C Threads does not support fixed-priority scheduling. We extended RT Mach with new kernel-level synchronization primitives that implement priority inheritance. These primitives avoid spin locks, and if a high-priority thread is waiting on a low-priority thread, the low-priority thread inherits the priority of the waiter until the lock is released. (More recent releases of RT Mach now have synchronization primitives with priority inheritance.)

- Scheduling Quantum. RT Mach will run a high-priority thread as soon as it is enabled to run rather than waiting for the next scheduling quantum as in Mach 3.0. This scheduling feature has recently been incorporated into Mach 3.0.

Among the remaining problems in RT Mach are:

- Nonpreemptive Kernel. There are still long path lengths in the kernel. Since the kernel runs without preemption, the kernel can create long delays. We feel that nonpreemptive kernels are not intrinsically bad, but one must be very careful to place time-bounds on all kernel calls and keep longer-running operations out of the kernel. In general, it seems that monolithic Unix kernels that have been made preemptable have the edge over microkernels, although this may be due to the fact that monolithic kernels are supported as commercial products, while microkernels are just entering the commercial world.

- Device drivers. While RT Mach fixes many of the problems of scheduling user-level threads, it does not include a rewrite of problematic device drivers. Since interrupt processing preempts user-level threads, the time spent in interrupt handlers and in AST's (kernel-level software event handlers) determines a lower bound on the latency to activate a user-level thread.

- Interprocess Communications. The kernel supported IPC facility is not designed for real-time communication. In particular, IPC can require dynamic storage allocation and paging.

Given that RT Mach is a research operating system, it is not fair to criticize RT Mach for these problems. In fact, the RT Mach project can take credit for demonstrating solutions to many real-time problems. It is especially gratifying to see these solutions making their way into other systems. It is also the case that the remaining problems in RT Mach are the subject of ongoing work. Furthermore, there may be some important differences between the goals of RT Mach and the assumptions of this study. We assume that conventional operating system support is essential, and that more timesharing-oriented support must coexist with multimedia applications. RT Mach and many commercial real-time kernels offer non-Unix interfaces or subsets of Unix in order to enhance real-time performance.

All caveats aside, our goal is to share practical experience and real performance measurements. We would like to stress again that the bottom line for real-time performance is the worst case. If nothing else, our measurements should serve as reminders of this fact and encourage designers to address all of the problem areas rather than a few.

2. Related Work

Our previous work [Dannenberg 93a] measured the latency and IPC performance of Mach 3.0. The cited report describes other related work, including work at Fujitsu Laboratories using Mach 2.5 [Nakajima 91] and Mach 3.0 [Nakajima 92] with real-time extensions similar to those of RT Mach. A recent paper describes synchronization primitives for RT Mach [Nakajima 93]. This work would have been ideal for our applications, but it was not implemented until after our own simpler synchronization primitives. Later, when the RT Mach group released a system that supported synchronization with priority inheritance, we adopted their code.

3. Latency Measurements

From our Multimedia Testbed, we ported an application that outputs 100 MIDI messages per second. The details of this application and the drivers that support it were reported previously [Dannenberg 93a]. To summarize, a user-level thread waits for a message from a timer device. When the message arrives, the user-level thread transfers data to an output buffer and awakens an output thread. The output thread writes directly to a hardware output port, and the data appears as serial output. The messages should be spaced exactly 10ms apart.

We measure the actual spacing to within 0.7ms using an IBM RS/6000 with our MIDI device driver that timestamps data within a high-priority interrupt. Note that since we measure the interarrival times of messages, we are not actually measuring latency, but jitter. In other words, we measure the variation in latency from one message to the next. We know from IPC and context switch measurements that the minimum latency is on the order of 1ms, which is negligible compared to the worst-case jitter.

On an otherwise unloaded system, output intervals fell mostly in the 9.5 to 10.5 millisecond range, with a standard deviation of 0.19ms, and a worst case observed of 10.7ms. Note that the reported times are actual inter-message times. Since the intended interval is 10ms, a measurement of 10.5 indicates a 0.5ms latency somewhere in the system. These figures are all within measurement error, so they do not actually measure latency. We can only conclude that latency is low. This measurement was performed running stand-alone with no network connection. When the network is connected, the worst-case time increases to 12.6ms. This still compares favorably to the 16.7 milliseconds observed with fixed priorities under Mach 3.0, and 21.6ms under Mach 3.0 without fixed priorities.

One must be careful in interpreting these numbers. When we observe a large delay, we can be confident that there is a problem. Thus, observations give us a lower bound on how bad things are. If we do not observe a long delay, we cannot assume that problems do not exist, only that we were not ‘‘lucky’’ enough to observe them.

To learn more about the worst case, we performed several experiments to stress the system in various ways. Our next test was to create many tasks and threads at once, which would run for a very short time and then destroy themselves. Surprisingly, this turned out to yield our worst observed interarrival time of 288.6 milliseconds. This indicates that non-preemption in the kernel is a serious problem. For this test, the network *was* connected, but there is evidence (see below) that the delays are not related to the network.

We next looked at disk activity as a source of latency. To obtain disk measurements, the network was disconnected. Writing to the disk yielded a worst case of 13.6ms, indicating that disk writes probably have little effect on latency. Reading the disk, on the other hand, led to a worst-case time of 30.8ms.

The network is a source of high latency. The worst-case delay observed was 190.8ms when another process was sending many network packets. (It was because of this high delay that we disabled the network for the test with disk IO.)

These numbers are no better than the previous measurements made using Mach 3.0 with fixed priority threads. For example, the maximum delay measured using Mach 3.0 was 149.6ms. We were more creative in stressing RT Mach, so it is likely that we did not find the worst case for Mach 3.0. Nevertheless, it seems that real-time performance is limited more by networking code and thread creation than by user-thread scheduling. It is disappointing that RT Threads did not provide a quick fix to the known problem of high-priority threads in the Unix server.

4. Interrupt Latency

Network and disk device drivers disable interrupts during critical sections to prevent preemption. Since we observed rather severe interference of real-time processing by these devices, we wondered what was happening at the interrupt level. Our next step was to connect a logic analyzer to the ISA bus, and measure the amount of latency between when an interrupt line is asserted and when the interrupt handler for that device is entered. To do this, we wrote a custom interrupt handler which output ff_{16} followed by 00_{16} directly to the parallel port output register each time it was called. This generates an easy-to-find logic transition when the interrupt handler gains control. The logic analyzer was configured to watch the interrupt line, and one of the pins on the parallel port, and recorded the times at which they went high and low. With this setup, we repeated those tests above.

Our baseline measurement had a mean latency of 24.03us (microseconds), a maximum latency of 29.58us, and a standard deviation of 0.244us. We repeated this test with lots of keyboard activity, and noticed no significant change in the response times.

Heavy disk activity yielded a maximum latency of 224.90us, a dramatic increase from the baseline. Intensive task and thread creation and destruction gave a maximum latency of 286.80us. Finally, we enabled the network and repeated the test with an intensive stream of network packets being sent to the target machine. The maximum measured latency was 4321.00us, or 4.321ms.

Given that MIDI data arrives at intervals as low as 320us/byte, a MIDI interface would need to capture, timestamp, and buffer at least 14 bytes to avoid lost data (which in MIDI is non-recoverable). A typical serial interface chip that buffers only a few bytes would be inadequate for MIDI under these conditions.

Another area where interrupt timing is critical, at least on Intel platforms, is with popular digital audio peripherals. Many of these expect the system to start up a DMA transfer within two audio sample times, so response times below 45us are required for 44.1KHz sample rates. It may be that this problem can be solved by choosing interrupt priorities carefully, but it is at least worrisome to imagine one hardware device (audio) requiring a response time of less than 45us, while another (network) is preempting threads for over 4000us.

On the same machine running DOS 5.0, we measured interrupt latencies in the 12 to 14us range. We did not have networks running, so this number is not directly comparable to the Mach measurements, but we note that the largest time we saw with DOS was about half of the smallest time we observed in Mach.

These numbers demonstrate clearly the impact that overly large critical sections can have on real-time performance. The network driver introduces some especially large latencies into interrupt processing time.

5. Digital Audio Application

Our digital audio application consists of a simple audio signal synthesis process controlled by MIDI input. The synthesis technique used is the Karplus-Strong plucked string algorithm [Karplus 83], in which a table of samples is repeatedly read and output. As samples are output, adjacent samples are averaged and rewritten to the table. The averaging operation is effectively a low-pass filter, so on each pass the table is slightly different. As with a real plucked string, the high-frequency harmonics decay rapidly, and the low-frequency harmonics decay slowly. This results in an interesting musical tone with very little computation.

Our synthesis computes only one note at a time. When a MIDI note-on message arrives, we compute a table size according to the MIDI key number, initialize a table with random sample values, and begin outputting samples in small blocks. We wanted to minimize the delay from incoming MIDI events to outgoing notes. Commercial synthesizers typically respond in fewer than 5ms, which corresponds to only 110 sample times or 220 bytes in our application. While efficiency concerns suggest that audio data be computed in larger blocks, this would negatively impact the latency of the application. To achieve a delay of 5ms, one must allow for input buffering and processing as well as output buffering, so blocks representing only a millisecond or two must be used.

In our application, computed samples are sent to an audio output server. The server appends incoming blocks to output buffers from which samples are transferred to an audio card by DMA. We generate 16-bit samples at a 22.05KHz sample rate, so the data rate is only 44.1KB/s.

Even at this data rate, and when computing a block every 10ms, we cannot achieve satisfactory performance. If output from the synthesis process is disabled, there is no problem computing the data in real time. Similarly, we can find no evidence that the audio server is having problems copying data or servicing DMA requests. However, when the output is sent via Mach IPC messages from the application to the audio server, we experience occasional long delays. This causes the output to underflow and fall behind. The application then gets ahead of the output, with data being queued in the form of a backlog of IPC messages. A growing message queue seems to make performance even worse. We were unable to obtain detailed timing of operations within the kernel, but evidently, the IPC operations are not deterministic enough for our application.

Newer versions of RT Mach offer a new integrated IPC and scheduling mechanism. Part of the change is meant to address the known indeterminacy in Mach IPC. For example, message buffers can be pre-allocated in the new system.

From this study, it is difficult to draw any definite conclusions. It seems clear that there are problems with Mach IPC and that these problems will have an impact on continuous audio and video applications. It also seems that more effort should be placed on instrumentation. It should be simple to instrument operations in the kernel, and it is surprising that a real-time research kernel does not make this easy.

6. Simple Scheduling Task

One of our applications is Tactus, a system for synchronized interactive multimedia. At the heart of the Tactus server (and many other multimedia presentation systems) is a process that transfers data from an input queue to devices according to timestamps in the data. This requires accurate timing in the presence of communication between multiple tasks. To simplify measurements, we made an experimental model of the sort of scheduler used by Tactus, and we subjected it to various controlled experiments.

Our goal is to assess the contribution of a real-time operating system to the performance of this scheduling task. The specific task is for a client process to send timestamped messages to a server process, which executes a procedure for each message at the time indicated by the timestamp. Each dependent variable measured is a lag, the difference between the time something actually happened and the time that thing was supposed to happen, as indicated by the timestamp.

The heart of the server is a thread known as the dispatch thread, which executes the following loop:

```
for(;;) {
    sleep_until(time of first message in queue)
                    [dispatchtimer]
    while (later than time of first message in queue) {
        remove the first message from the queue
                    [before]
        dispatch it (call the associated function)
        free the storage associated with the message
                    [after]
    }
}
```

The client, indirectly through another thread in the server, is enqueing messages on the queue. When a new message is enqueued, the sleep will return immediately, thus the ‘‘later than’’ test above is essential. In the case of an empty queue, the argument to `sleep_until` is made very large.

The `sleep_until` function is performed via `mach_msg`, waiting for a message on a port with the appropriate timeout. The timer may thus be easily interrupted by actually sending a message to the appropriate port. Since the timeout on `mach_msg` has a resolution of 10ms, this places a lower bound of 10ms on the accuracy of our measurements (in other words, times are quantized to 10ms). Better timing resolution can be obtained by using multiple threads, the timer facility, and synchronization primitives, but this is awkward. Another option would be to use `rt_mach_msg`, but this call was not working at the time we performed our tests.

The lags measured are the difference between the timestamp given in the argument of `sleep_until` and the actual time the various bracketed words in the above pseudocode are reached. The *dispatchtimer* lag measures the delay of the operating system in waking up the sleeping thread on time. The *before* lag measures lateness in the time the desired function is called, and the *after* lag measures the total delay, including calling the desired function and freeing resources associated with the message. All lags are measured in milliseconds.

There are a number of independent variables of interest:

C Thread or *RT Thread*
: In the *C Thread* case, server and client threads are all implemented by the Mach 3.0 C Thread package, with all the threads being scheduled by the default timesharing scheduler, and the client ‘‘niced’’ to run at a lower priority than the server. In the *RT Thread* case, the client and server threads are all fixed-priority real-time threads running under RT Mach, with the dispatch thread running at the highest priority.

Inter-Arrival Time (IATime)
: This is an integer representing (in milliseconds) the period at which messages are to be dispatched from the server. E.g. an *IATime* of 5 means a message should be dispatched every five milliseconds, or 200 per second. An *IATime* of 100 is 10 messages per second. Thus, *IATime* is be coarsely considered to be inversely proportional to utilization.

Load or *No Load*
: In the *No Load* case, the client and server are running on an unloaded machine (though still one connected to the network). The *Load* case represents running the system along with a heavy I/O load. The command:

```
repeat 1000 cat /usr1/pagingfile > /dev/null
```

generates the load. The command runs at Unix scheduling priority, so theoretically the fixed-priority threads of RT Mach should be preferred by the scheduler. It turns out that there is synchronization between the `cat` thread and the paging threads in the Mach kernel since we are reading the paging file.

Smallseg or *Bigseg* Messages from client to server come in segments, which are fixed-sized blocks of shared memory. The default is to have segments big enough to hold an entire frame of video (20Kbytes), but in this test smaller segments (512bytes) may also be used. In *Smallseg*, multiple segments fit in a page; in *Bigseg* the reverse is true. Presumably, bigger segments cause more paging.

Generally, we looked at the effect on average and worst-case lags at various utilizations when changing one variable.

6.1. RT Threads vs. C Threads, No Load

To start, consider the *dispatchtimer* lag on an unloaded system using small segments, RT Threads vs. C Threads:

	Average Lag		Worst-Case Lag	
IATime	RT Thread	C Thread	RT Thread	C Thread
5	6.4	5.7	55.5	7.7
7.5	5.6	6.4	9.3	11.9
10	6.0	4.2	6.8	6.1
12.5	4.6	4.1	8.7	10.4
15	5.7	4.5	8.7	9.1
20	2.2	5.8	2.8	8.0
25	5.0	4.7	48.5	9.1
30	7.8	7.9	8.3	10.8
35	4.7	5.7	8.1	11.0
40	3.8	5.5	4.3	7.6
45	4.0	6.8	7.5	12.1
50	7.2	8.0	10.3	9.7
75	2.4	6.7	22.9	20.6
100	3.6	3.0	3.9	4.4
150	4.2	2.9	4.5	4.6
200	8.1	8.7	8.4	9.1

In this case, there is little difference between C Threads and RT Threads. There are occasionally very long delays in the RT Threads case. We do not know the source of these delays.

6.2. RT Threads vs. C Threads, Under Load

Performing the same experiment with a background load, the results are much different. Here, the superiority of RT Mach is apparent.

IATime	Average Lag RT Thread	Average Lag C Thread	Worst-Case Lag RT Thread	Worst-Case Lag C Thread
5	6.7	8.8	8.3	44.3
7.5	5.9	7.0	10.4	17.4
10	6.1	4.8	7.7	40.8
12.5	5.2	6.3	10.1	18.6
15	5.9	5.9	10.2	20.0
20	2.1	5.3	3.8	13.2
25	6.3	6.2	10.3	18.7
30	1.2	5.3	11.4	18.6
35	5.4	5.2	9.9	17.0
40	5.7	7.1	7.3	19.9
45	3.4	5.9	11.4	20.1
50	2.6	2.6	4.2	14.5
75	6.5	4.9	11.0	14.9
100	3.3	7.1	4.9	17.3
150	3.5	7.0	4.9	13.8
200	2.7	7.3	4.2	14.8

Note that the worst-case times for RT Threads are better under load than not under load for some cases (compare this table to that in Section 6.1). Perhaps this is due to a kernel bug.

6.3. Segment Size

In our tests, there is a slight increase in worst-case lag as the segment size increases, but the data is not conclusive. This is the true for both RT Threads and C Threads, and for the Load and No Load cases, so we will not present the data here.

6.4. Before and After Lags

The *before* lag measures the lateness in the time the dispatch function is called. The *after* lag measures the time to call the dispatch function, execute the dispatch function, and reclaim the associated message memory. The results of comparing C Threads to RT Threads on the basis of *before* lags and *after* lags are more or less identical to results for the *dispatchtimer* lags. This might be expected, because we are simply measuring a small amount of additional work with these lags.

6.5. Discussion

There is some evidence that RT Threads provide superior performance over C Threads, but it is not as obvious as one might hope. Priority scheduling is good, but paging must be considered as well. Mach allows the wiring down of certain pages to physical memory. It is easy to wire down all data containing messages. More difficult, but still possible, is wiring down all code potentially callable by the dispatch thread. This is work we should do to better assess the potential of RT Mach.

Another potential improvement would be to do less work between the time the `sleep_until` finishes and the dispatch occurs. Currently, a lock is obtained and the first message is dequeued. At the cost of slightly more complexity, the first message can be dequeued before the `sleep_until` returns.

However, paging cannot account for some awful worst-case performance of the use of timeouts in `mach_msg` as timers. The only explanation seems to be an error or delay in the RT Mach fixed-priority scheduler. Delays of about 50ms were observed. There seems to be a flurry of Mach scheduler rewriting by various groups, so it is reasonable to expect this anomalous behavior to disappear in future versions of Mach.

7. Software Digital Video

We used Tactus to implement a software digital video playback system. Since we have reported on the features of Tactus and its support for interactivity elsewhere [Dannenberg 93b, Rubine 94], we will restrict this discussion to the performance of Tactus under RT Mach.

The software digital video task is related to the previous scheduling task. However, now the dispatch thread will be used to send a message to the X server, which (in the RT thread case) is running a higher priority than the dispatch thread. The effect should be a handoff, where dispatch thread's message immediately wakes up the X server, which draws an image. This is exactly the effect one would obtain in a system with a more elaborate priority hand-off from client to server. Shared memory is used between the client and Tactus server, and also between the Tactus server and the X server, but a copy occurs at dispatch time. Each frame is approximately 20K, and the copy takes about 2ms, excluding any paging required.

The video contains 217 frames. Two messages are queued at the Tactus server for each frame: the first contains the frame, and the second contains the flush command that forces the frame to the X server.

The entire system consists of the client, the Tactus server, and the X server. The client reads frames from the disk and sends them to the Tactus server, which buffers them and sends them to the X server. When real-time threads are used, the X server has the highest priority, Tactus server second highest, client least. All tests were run on an otherwise unloaded machine, with the exception of network and other incidental processes.

There are two independent variables: Rate and Thread. Rate is the number of frames per second to be drawn (we used rates of 5, 8, 10, and 12 fps), and Thread indicates whether RT Threads or C Threads were used in all three processes. (Note that we constructed a special version of X11 using RT Threads for this experiment.)

Before running tests for a given rate/thread pair, we rebooted the machine and started the three processes. Once started, each movie was run at least five times. We threw away any runs with glitches. For a given rate/thread pair, we would expect the later runs to have less paging.

The measured variables include all the dependent variables mentioned in the previous test, as well as additional measurements made in the X server. In particular, we measured *iabefore*, the interarrival times at a point just before a frame was drawn in the X server, *iaafter*, the interarrival times just after a frame was drawn, and *drawtime*, the time it took to draw a frame. These times are accurate to 1ms.

If we plot histograms of the draw times, we can see clearly that the draw times with RT Threads have less variance than C Threads. Figures 1 to 4 compare RT Threads to C Threads at different frame rates. The histograms combine data from 5 runs of 217 frames and show the frequency of occurrence of draw times in 1ms steps. Note that the frequency (vertical) axis is plotted on a logarithmic scale to emphasize the relatively few occurrences of poor performance. In our data, we replaced zeros with 0.6 since the logarithm of zero is undefined.

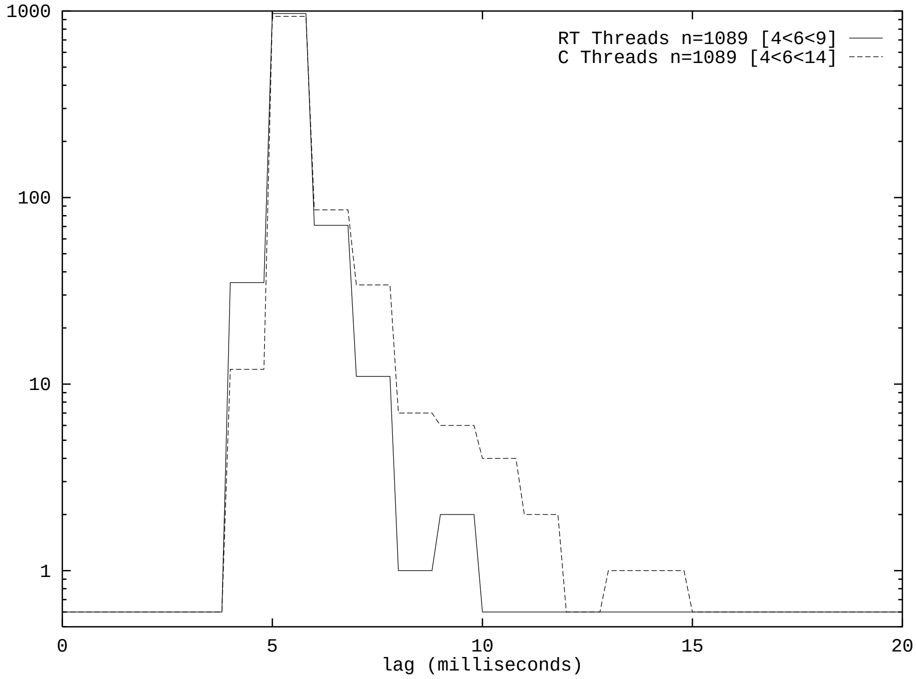

Figure 1: Histogram of frame draw times using X servers based on RT Threads and C Threads, drawing 5 frames per second.

We see for both C Threads and RT Threads, it takes on average 6ms to draw a frame, with 4.9ms being the minimum frame draw time. However, using C Threads leads to a much worse maximum draw time — generally 10 to 20ms, and sometimes as bad as 69ms. This is compared to a maximum draw time of 7 to 10ms using RT Threads. The difference is presumably due to the fact that the C Threads X server may be preempted during a draw. The RT Threads X server, being the highest priority thread in the system, will not be preempted as long. However, paging is possible in either case, giving rise to maximum times significantly larger than average for both RT and C Threads.

7.1. Inter-frame Periods

Now we will look at inter-frame periods, the time between successive frames. Note that the average interarrival time depends on the frame rate as expected. We look at the ‘‘before’’ times — the ‘‘after’’ times are similar. Times for the 5 and 10 frame per second runs are plotted in histograms in Figures 5 and 6.

In these plots, there is a surprising amount of variance in the RT Threads case as well as in the C Threads case. Although it appears as though long inter-frame periods are slightly more common with C Threads, RT Threads do not give uniformly better performance. Part of the explanation for long delays is paging, and often the first run at each frame rate had a large delay, presumably due to paging. We tried running this application with virtual pages wired to physical

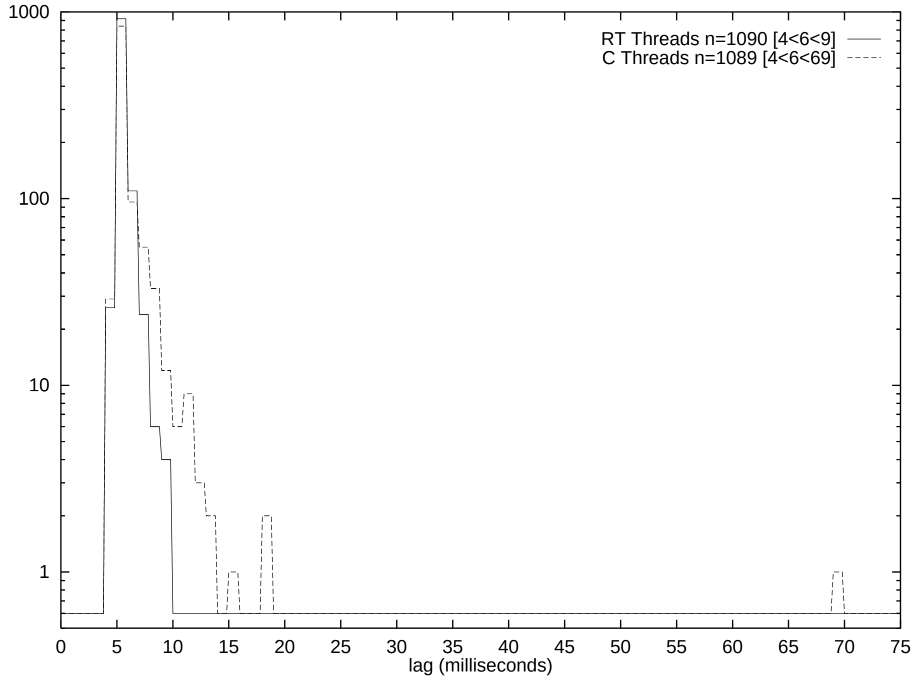

Figure 2: Histogram of frame draw times using X servers based on RT Threads and C Threads, drawing 8 frames per second. Note that the horizontal scale differs from Figures 1, 3, and 4.

memory, but this resulted in *worse* behavior due to thrashing elsewhere in the system.

8. Summary

This work follows a previous study of Mach 3.0, in which we found a number of barriers to achieving predictable performance in multimedia applications. One of the goals of RT Mach is to overcome these barriers and deliver predictable performance to real-time applications. In this study, we ported our applications to investigate their performance with support from RT Mach. In general, RT Mach was effective in making performance *more* predictable, but the level of predictability was *not sufficient*, in our opinion, for a wide range of multimedia applications.

As with our Mach 3.0 study, these results are not too surprising in retrospect. While RT Mach does offer a real-time thread facility, it is still dependent upon a kernel with long path lengths, some of which are unbounded, a message system that uses virtual memory, and an interrupt and device IO structure that preempts even real-time threads for long periods of time.

One of our measurements was a simple test of the latency from a hardware interrupt to a user-level process and back out to a device. We found the worst-case latency to be almost 300ms, due to long non-preemptable paths in the microkernel. We also found long latencies due to networking (190ms) and disk IO (30ms).

Another study looked at latency from interrupts to interrupt handlers. Using a MIDI interface,

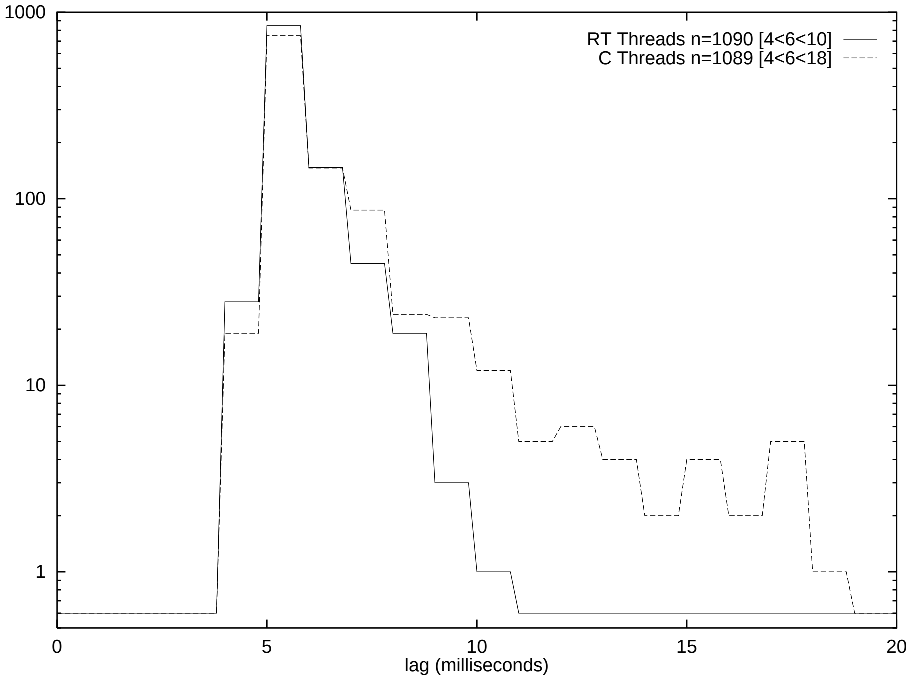

Figure 3: Histogram of frame draw times using X servers based on RT Threads and C Threads, drawing 10 frames per second.

we measured a latency of 4.3ms (milliseconds, not microseconds) when networking activity was present. The latency due to interrupt handlers alone would create problems for some applications and even for other devices.

Another set of measurements looked at the ability of an RT Mach thread to perform a task at a specified time, such as to deliver a video frame to a display server. This can be seen as a variation on the latency measurements reported above, except that the task is performed by a server that receives timed commands or data from a client. When the machine is lightly loaded, RT Mach performs well, and the performance is generally superior to that of Mach 3.0. When the system is more heavily loaded, significant latencies up to 200ms are observered.

Some of the observed latency in this case seem to be due to paging. With our large application, we found that pinning all of the pages of the X11 server in main memory led to even worse performance because of increased paging in other parts of the system.

9. Conclusions

There has been some debate about the importance of real-time support in operating systems. Our work will not resolve this debate, and in fact, it can be interpreted in two ways. On the ‘‘pro-real-time’’ side, our work shows that without real-time support, operating systems are not going to provide predictable real-time performance. It is true that machines are getting faster, but we are orders of magnitude away from the millisecond-level requirements of many applications. On the ‘‘con’’ side, our measurements show that there is little difference between

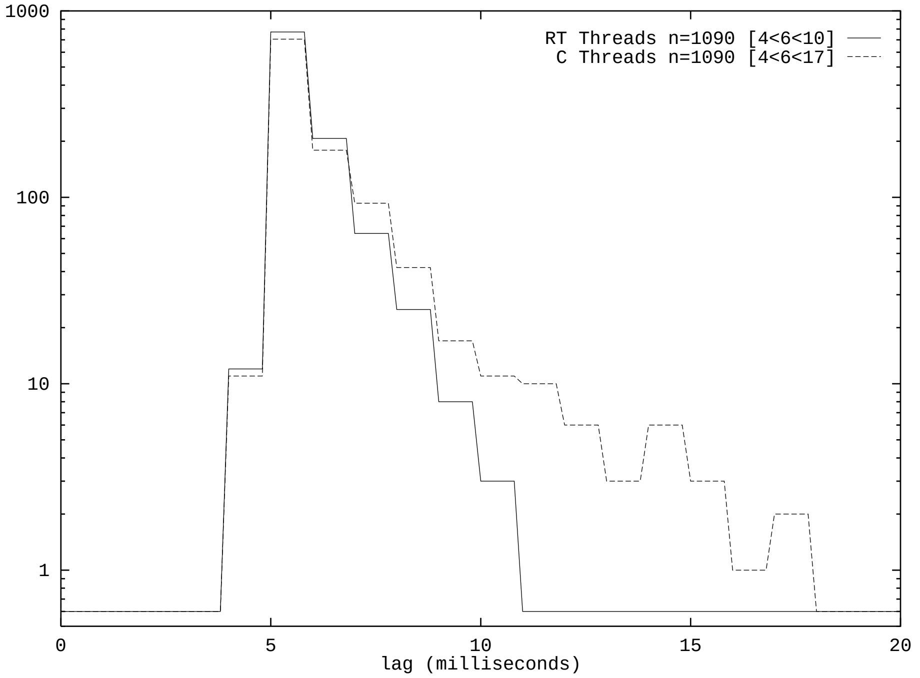

Figure 4: Histogram of frame draw times using X servers based on RT Threads and C Threads, drawing 12 frames per second.

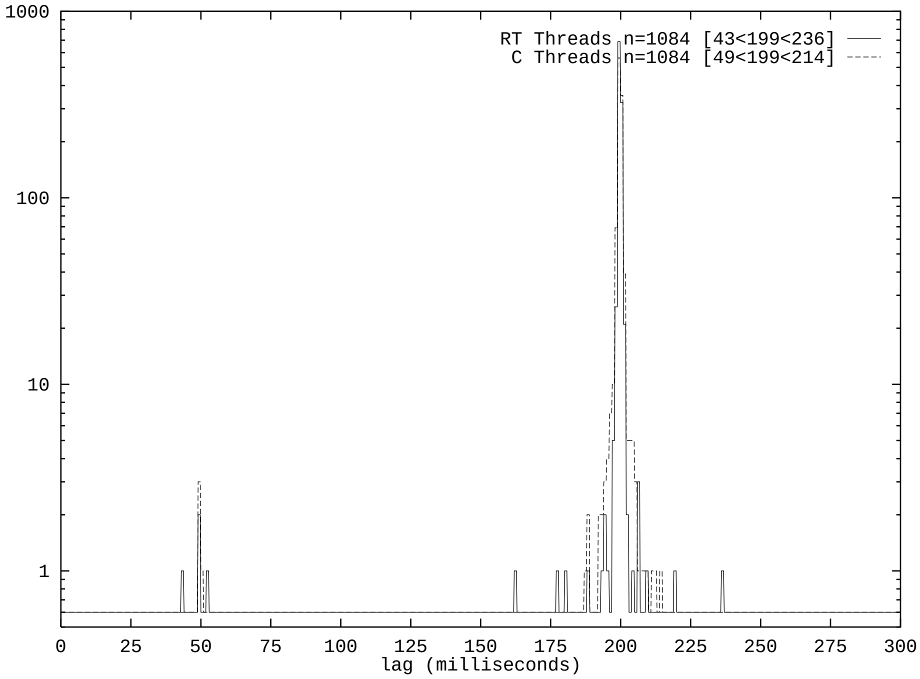

Figure 5: Histogram of inter-frame periods using X servers based on RT Threads and C Threads, drawing 5 frames per second.

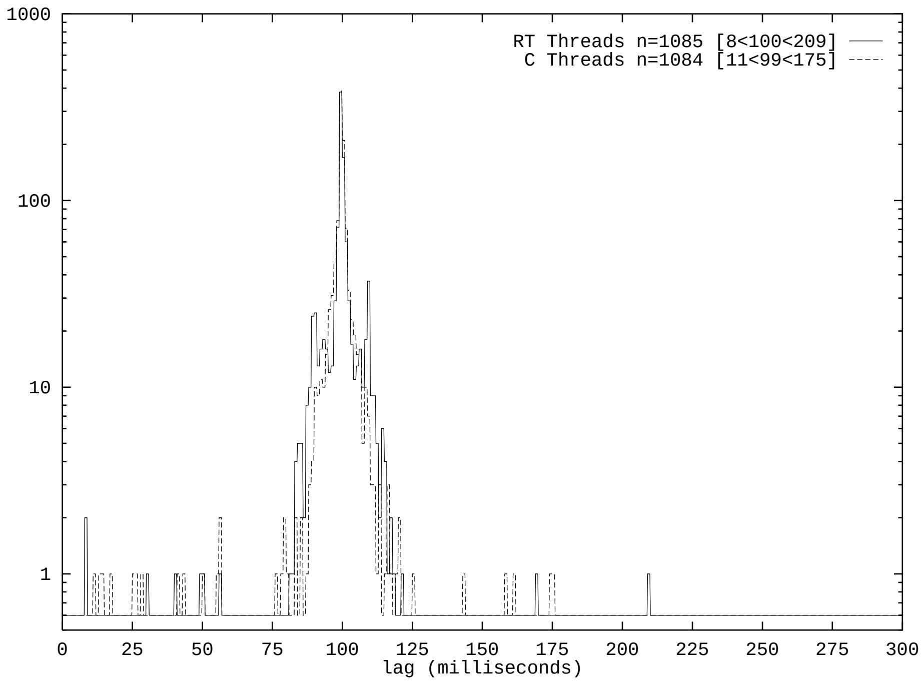

Figure 6: Histogram of inter-frame periods using X servers based on RT Threads and C Threads, drawing 10 frames per second.

the performance of RT Mach and ordinary Mach on lightly loaded machines, and on loaded machines, neither system delivers adequate performance. At least for our applications, RT Mach provides very little practical benefit.

One point we would like to make is that systems designers do not generally evaluate how well designs support predictable performance. The sum total of many small design decisions have a large aggregate impact on systems. Even when a large effort is made to correct these design problems, as in the case of the RT Mach project, the task is not straightforward. An example is the X11 server, which is a large piece of software, developed in a virtual memory, non-real-time environment. X11 is full of non-real-time features, and its very size makes it extremely difficult to use in a real-time system.

On the question of paging, we found that the X server caused particular problems. Wiring down the entire X server takes too much memory, but not wiring any pages leads to paging delays. It would be desirable to lock only the parts of the X server (and other system code pages) that are hit in time-critical paths such as video playback. One can imagine a kernel primitive in which a begin call is made, then the system runs for a while, and an end call is made, resulting in the wiring of pages touched during the run. This might enable the working set to be dynamically determined rather easily. In a multi-threaded X server, parts of the server could page without a major impact on the time-critical threads.

The RT Mach kernel is based closely on the Mach 3.0 kernel, and there are many problems that interfere with real-time performance. Some of these problems have been addressed, but others have not. We note again that work on RT Mach is still in progress, and at least some of the remaining problems will be addressed. In the meantime, real time threads seem to work, but their benefit is mitigated due to delays from other sources, including interrupts, the IO system, paging, and interprocess communication.

Recent work in RT Mach has gone into integrating communication with scheduling so that a client's priority can be handed off to the server [Nakajima 93]. Our video experiments indicate that this effort will not offer any real improvements for our applications, since our priorities are already correct. In other words, the problem is not with priority choice or scheduling policy, but with other factors.

In spite of our negative tone so far, we believe that real-time operating systems are essential, and the problem is that RT Mach does not go far enough. The developers of RT Mach are aware of its limitations, and have remedies for them. Also, to be fair, we should note that RT Mach has never been advertised as a finished solution to real-time problems. Our work indicates that a complete solution will require careful design not only of the kernel and device drivers, but also servers that provide for graphics display, operating system facilities, file IO, and network access. Systems that take a more comprehensive approach to supporting real-time applications will deliver significantly better multimedia performance than current systems.

10. Acknowledgments

This research was performed by the Carnegie Mellon Information Technology Center and supported by the IBM Corporation. This work would not have been possible without help from the CMU Mach Project, the CMU Advanced Real-Time Systems Project, and the CMU CS Facilities group, all of which helped us to get our systems running. Jim Zelenka worked on our project on a part-time basis, and played an important software development role in this work.

References

[Accetta 86] Accetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of Summer Usenix*. Usenix, July, 1986.

[Dannenberg 93a] Dannenberg, R. B., D. B. Anderson, T. Neuendorffer, D. Rubine, J. Zelenka. Performance Measurements of the Multimedia Testbed on Mach 3.0: Experience Writing Real-Time Device Drivers, Servers, and Applications.Technical Report CMU-CS-93-205, School of Computer Science, Carnegie Mellon University, July, 1993.

[Dannenberg 93b] Dannenberg, R. B., T. Neuendorffer, J. M. Newcomer, D. Rubine, D. Anderson. Tactus: Toolkit-level Support for Synchronized Interactive Multimedia. *Multimedia Systems Journal* 1(2):77-86, 1993.

[Karplus 83] Karplus, K., and Strong, A. Digital Synthesis of Plucked-String and Drum Timbres. *Computer Music Journal* 7(2):43-55, Summer, 1983.

[Nakajima 91] Nakajima, J., M. Yazaki, and H. Matsumoto. Multimedia/Realtime Extensions for the Mach Operating System. In *Proceedings of the Summer USENIX Conference*, pages 183-198. USENIX Association, 1991.

[Nakajima 92] Nakajima, J., M. Yazaki, and H. Matsumoto. Multimedia/Realtime Extensions for Mach 3.0. In *Usenix Workshop Proceedings on Micro-kernels and Other Kernel Architectures*, pages 183-198. USENIX Association, 1992.

[Nakajima 93] Nakajima, T., T. Kitayama, H. Arakawa, and H. Tokuda. Integrated Management of Priority Inversion in Real-Time Mach. In *Proceedings of the Real-Time Systems Symposium*, pages 120-130. IEEE Computer Society, 1993.

[Rubine 94] Rubine, D., R. Dannenberg, D. Anderson, T. Neuendorffer. Low-Latency Interaction Through Choice-Points, Buffering, and Cuts in Tactus. In *Proceedings of the International Conference on Multimedia Computing and Systems*, pages 224-233. IEEE Computer Society, 1994.

Table of Contents

List of Figures