Performance Measurements of the Multimedia Testbed on Mach 3.0: Experience Writing Real-Time Device Drivers, Servers, and Applications

Roger B. Dannenberg, David B. Anderson, Tom Neuendorffer
Dean Rubine, Jim Zelenka
July 21, 1993
CMU-CS-93-205

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

Abstract

Multimedia has generated a widespread interest in real-time support within general purpose operating systems. In addition to real-time support, multimedia often requires large amounts of data to be moved between devices and processes. 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 the Mach 3.0 Operating System where performance was measured. Experiments were conducted to compare various techniques for IPC and disk reads. Experience with fixed priority scheduling, user-level device drivers, and a high-resolution timer are also reported.

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.

In recent years, multimedia has become a key concern in the architecture of operating systems and applications. Multimedia has many definitions, but from the operating system perspective, we will take ‘‘multimedia’’ to mean conventional graphical display capability augmented with continuous-time digital media such as animation, video, and audio. New system support is required for multimedia because multimedia presentations inherently take place in real time, multimedia data rates are typically higher than those for simple text or graphics, and system abstractions for time, synchronization, media access, and media processing offer new areas for system support.

In this report, we examine the Mach Microkernel [Accetta 86] as a base for multimedia support. Our method is to develop applications, often ported from other systems, and measure their performance. Unlike many performance studies that isolate and measure a single operation, our focus has been on ‘‘end-to-end’’ performance measurements in a system configured for normal use. In addition, we measured individual operations to explain the overall results.

Our measurements are guided by two principal concerns. The first involves response latency, and we measured this with a set of music programs that output MIDI signals to music synthesizers in response to user input or to timer interrupts. The second involves interprocess communication, and we implemented a digital video application to stress this aspect of the operating system. We did not explore real-time network communication because Mach has no real-time support in this area.

Several techniques were used to improve real-time performance of our applications. First, Mach supports fixed-priority threads which are scheduled before ordinary timesharing threads. Second, we used a kernel-based high-resolution timer service. Third, we used a ‘‘user-level’’ device driver to avoid Unix I/O. Finally, we used shared memory interfaces for efficient interprocess communication.

All 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 unless otherwise noted. 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.

1.1. What should we expect?

In many ways this paper is unfair to Mach and its designers. To run real-time programs on a non-real-time operating system is asking for trouble. It should come as no surprise that we found it. A number of noted researchers, however, believe that real-time problems can be handled by current systems such as Mach. Anecdotal evidence is often cited, such as the ability to handle network interfaces without losing packets, the fact that faster processors and memory are reducing context-switch and IPC times, and the fact that microkernels can reduce the time during which preemption is disabled or shared resources are locked. All these are valid points, but there are other factors to consider.

Mach is still a fairly Unix-oriented system, and applications ordinarily make many calls to the Unix server for services such as I/O and time. Since the Unix server has very unpredictable performance, we should not expect real-time performance from any of its clients. Another set of issues has to do with scheduling. 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.

Our goal is to share practical experience and real performance measurements. We would like OS designers to consider that latency and predictability should be good enough for multimedia even in the worst case. This is very different from almost always good enough. This report may shed some light on the extent to which work is required for multimedia support. Finally, it is important to make things easy where possible. Our experience may encourage designers to make systems easier to use.

At the time of this writing, Mach is being modified to incorporate facilities from Real-Time Mach (RT Mach [Tokuda 90]), and we are porting our applications to RT Mach. We expect better real-time performance and interesting new results in the future. By studying Mach now, however, we can obtain benchmarks that can be compared to RT Mach later, and we can estimate what kind of performance can be reasonably expected.

The next section describes related work. Sections 3 through 6 describe operating system modifications and device driver issues associated with our set of MIDI (music) applications. Sections 7 through 11 then describe system issues and measurements for our digital video applications. Section 12 presents our conclusions.

2. Related Work.

Researchers at Fujitsu Laboratories have used Mach for multimedia applications. Mach 2.5 was extended with real-time threads and other facilities to support multimedia applications. [Nakajima 91] The main parameter studied was event dispatch latency. A worst case of about 20ms was achieved by a combination of scheduling and support for preemption in the kernel.

Subsequent work [Nakajima 92] produced even better performance under an extended Mach 3.0, where worst-case latencies of about 10ms were observed. An analysis of the sources of latency points to two problems: the SCSI disk interrupt handler can cause unpredictable delays and interrupts can interfere with real-time threads.

These two studies both rely on kernel extensions similar to those provided by RT Mach, so we are encouraged by the low latencies reported. These studies did not measure IPC performance, disk performance, or run with heavy network traffic, so there are remaining issues to consider.

The user-level I/O facilities of Mach that are used in our study are described in [Forin 91]. This work is extended in [Golub 93]. A shared memory interface to XII and its performance is described in [Ginsberg 93]. Our XII server is a descendent of this work.

3. Low-latency applications.

A real-time system that deals with temporal media should be able to provide low-latency response to user input. An extreme case is a music performance system where input from a music keyboard must be routed to a synthesizer within a few milliseconds. Another example is a video or audio editing system where a user might tap the space bar to mark an edit point while watching or listening in real time. User input should be captured and timestamped within about 10ms.

Our original application was a music conducting system in which the user taps a tempo and the conducting system dynamically adapts to the user's tempo. In this application, a musical score is loaded from disk to memory before the performance. Once the performance is started, input comes from the user's taps and from the real-time clock, and output is to the MIDI serial port, which is connected to an external music synthesizer. One of the interesting characteristics of this application is that input handling must be done in real time by the application program. Although possible, it would be unusual to put this sort of complex, adaptive input handling in a device driver.

While this application is good to illustrate requirements, we soon learned that a much simpler system could exhibit the same basic requirement but make measurement easier to perform. We wrote a program that outputs 100 MIDI messages per second. This program is similar to the conducting program in that it requires low-latency response from input (the real-time clock device) to output (MIDI) and runs at the application/user level. In addition, the output in this case is simple and predictable, making it easy to measure performance. Before reporting on performance, we will describe the real-time clock and MIDI device drivers in some detail. Readers may wish to read Section 4.4 and then skip to Section 5.

3.1. Providing real-time clock services.

The real-time clock services were provided by integrating a modified version of Jack Test's clock service for OSF-1/MK into the MK69 version of Mach. (Mach now has a real-time clock in the standard release.) The goal of this clock model is to provide accurate services for setting and reading the clock, along with an alarm service for timing events.

Time is represented by a `timespec` — a structure containing two integers. The `timespec` is P1003.4 compliant; that is, the first value represents a number of seconds, the second a number of nanoseconds (valid in the range $[0,10^9)$). The system real-time clock is a `timespec` representing elapsed time since the system boot.

The alarm service is provided in two forms. In each, times may be expressed either as an absolute time since the last reboot, or as a time measured from when the IPC is received by the kernel. The first form acts very much like the normal Unix `sleep()` call: The calling thread is suspended until the specified time is reached. The second is asynchronous: An IPC is sent to a specified port when the time is reached. The granularity of the timing of these alarms is adjustable; values as small as half a millisecond have been used without adverse effect upon system performance (see Section 6 for measurements).

The in-kernel implementation of this clock service is relatively straightforward. The granularity of the clock operations is mapped into a frequency, which is used to determine how often the system clock is updated, and how often alarms are checked. Whenever a thread sets an alarm (synchronous or asynchronous), it is placed in an ordered list of alarms. Alarms are checked within the clock interrupt. If the alarm was synchronous (the thread is suspended until the time of the alarm), the thread is flagged as ready to run, and will run according to its priority when the scheduler schedules it. If the alarm was asynchronous (IPC), it is placed upon another list, this one composed entirely of expired alarms. At this time, another kernel thread, the alarm thread, is marked as ready to run at high priority. When this thread runs, it traverses the list of expired alarms, removing them from the list and sending an IPC to each port listed. The reason that the IPCs are not sent from within the clock interrupt is that interrupts may come at any time, even within other IPCs. Therefore, it is not necessarily safe to send an IPC during an interrupt. (An alternate faster approach is to create a software interrupt — known as an ‘‘AST’’ in Mach — which would run ahead of regular kernel threads, at a time when it is safe to send IPCs.)

The interface between user processes and the kernel clock services are provided by IPCs. Messages to the clock server are dispatched in the same manner as other kernel services (such as the device server, the norma server, etc.). Messages are sent on one of two ports — the clock port and the clock control port. Actions which change the state of the clock (such as setting the clock rate, the alarm resolution, etc.) can only be performed using the clock control port; normal actions (such as querying the time and setting alarms) can be performed with either port. There are IPCs to query both of these ports from the kernel. To get the control port, one must use the host privileged port.

4. The MIDI device driver.

Our MIDI device driver interfaces to a Roland MPU-401 or compatible interface. This interface has small buffers for incoming and outgoing data. Interestingly, the MPU-401 will interrupt the host when the input buffer becomes non-empty, but there is no interrupt when the output buffer becomes non-full. We use the MPU-401 in ‘‘UART Mode,’’ which means both input and output pass through the device as uninterpreted byte streams.

4.1. Overview.

The MIDI device driver is divided into three major components: the user-level output portion, the in-kernel driver, and a user-level input server. Because output to the MPU-401 MIDI interface is accomplished by polling a register to determine whether or not the adapter is ready for data, and storing a byte in a register on the adapter when it is, no in-kernel code is necessary to handle interrupts, DMA, or other low-level hardware requirements for output. On the other hand, MIDI input requires that an interrupt be serviced when the MPU-401 wishes to present data to the system, and that this data somehow gets moved into user address space. The in-kernel portion of the MIDI driver does exactly this; when an interrupt arrives, it reads any available input bytes into a page of memory and raises a semaphore also located in this page. A user-level server maps this page into its own address space, and waits on the semaphore. When it awakens, it parses the new data into timestamped messages and sends them to its clients via IPCs.

4.2. MIDI Output.

Output is provided in the library `libmidi.a`, which provides these functions to other levels: read a data byte, read the status byte, send a data byte, and send a command byte. When asserting a byte (data or command), the caller is responsible for determining that the MPU buffer is not full by reading and checking the status byte.

A more generally useful level of functionality is provided by the real-time MIDI library (`libmtmidi.a`). This library provides user-level processes with the ITC standard MIDI interface (documented separately). When an `mi_id` (an abstract representation of the MPU used by the user process, and passed to all `mtmidi` functions) is created, several threads are automatically spawned and associated with the new `mi_id`.

One of these is a useless thread; it sleeps continuously. Its purpose is to receive signals. Because signals are a Unix-level concept, they do not exist as far as low-level Mach functions are concerned. Therefore, unlike normal Unix system calls, Mach system calls cannot be interrupted by signals. The behavior that results from this condition is somewhat undesirable; Mach programs in the midst of blocking while performing a Mach-specific operation (such as waiting for an IPC, as is most often the case within all Mach system calls) will not stop if the process is ^C'd, or receives any signal (except signal 9, which the Unix server does not deliver to the process but instead uses to abort the process's threads). A solution to this problem is to have a process blocking on a Unix-level function (in this case, `sleep()`). This thread will receive the signal, which will then have its normal effect on the running process.

Two threads work together for MIDI output. Using the ITC MIDI library, all outgoing messages have associated with them times at which they should be sent. When any thread in the ordinary user-process enqueues a message, a function in `libmtmidi` enters a critical section and inserts the message into a queue of outgoing messages (unless it's timestamp is such that it is late or out-of-order; handling of these messages will be explained below). After doing so, it asserts a condition (a C-Threads equivalent of a semaphore) that the queue has changed, and returns.

The first of the output threads is interested in this condition. When it receives it, it wakes up and examines the message at the head of its queue. It then uses the real-time kernel sleep service to wait for the time when the message should be dispatched. When it awakens after sleeping, it first examines its associated `mi_id` to see if it has been closed. If so, the thread exits cleanly. Otherwise, the thread continues to execute normally. All messages whose dispatch times have come are removed from this queue of outgoing messages and placed in another queue. This second queue contains nothing but messages that should be dispatched immediately. (When late or out-of-order messages are enqueued, they are placed directly into this second queue). The thread then asserts the condition that this second queue has changed, and re-examines the first output queue. If the queue is empty, it will wait for the condition that something has been placed in this queue, otherwise it will sleep until the dispatch time of the item at the head of the queue.

The second output thread actually does the work of sending the bytes. As long as there is any data in the queue of messages to be dispatched immediately, it will continue to send bytes. When this queue is empty, it will wait for the condition that something has been placed in this queue to be signalled. (This thread also examines its `mi_id` when it awakens to determine if it should exit). As long as the MPU-401 output buffer is not full, it will continue to send bytes. When the

MPU-401 is full, the thread will sleep for a few milliseconds to give the MPU-401 time to clear its buffer.

4.3. MIDI kernel input.

The original conception of the MIDI device driver was that it would use a page of memory shared between the kernel and the application process. The interrupt handler would write data into this shared page, and the user-level process would read from it. Inevitably, this raises problems of synchronization. Within the microkernel, there is a set of functions that provide semaphores explicitly to be used for synchronizing an interrupt handler's communications with a user-level process in a page of shared memory.

In the kernel version used for our work, the user-level process is required to delete semphores before exiting; otherwise, a system crash could occur. (This problem has been fixed in more recent releases.) This led us to implement the device driver as a server process rather than link the driver into the application process.

When the machine boots, before the Unix server is started, the microkernel initializes all its devices. First, it calls the probe method of all its drivers to determine what devices are present. The probe method indicates whether or not a device is present, and if it is, it allocates any necessary storage for dealing with this device. (This is where the page to be shared with the user-level process is allocated; this shared page must also be wired to avoid swapping). The device should not enable interrupts at this time. Later in the boot, the microkernel calls the attach method of each device found by probing. This method associates the driver with its interrupt.The open method is called when the user-level process performs a `device_open`. This method initializes the semaphore in the shared page, and resets the MPU-401 MIDI interface hardware. When the user-level process wishes to access the shared page, it calls `device_map`, which results in a call to the `mmap` method in the MIDI driver. This method must return a special indicator of the page to be shared. To do so, it maps the kernel virtual address for this page directly to a physical address (using `kvtophys()`), and then uses `i386_btop()` to map this physical address to a page indicator.

4.4. Device driver issues.

Let us summarize the design issues thus far. Traditional (at least for Unix) in-kernel device drivers offer efficiency and a variety of implementation options, but development is painful. This led Mach designers to add support for ‘‘user-level’’ device drivers [Forin 91]. Despite the name,‘‘user-level’’ drivers require in-kernel interrupt handlers that signal user tasks and perform time-critical data transfers (such as moving data from a register to a buffer). Since interrupt handlers cannot be loaded at run time, installing a new driver requires that a new kernel be built. (IBM has implemented dynamically loadable interrupt handlers in a proprietary version of Mach.) Drivers based on IPC are slow, and messages cannot be sent from within interrupts anyway, so the preferred structure for a ’’user-level’’ driver is a shared page for data transfer and a semaphore for asynchronous event notification.

4.5. MIDI input server.

When the MIDI input server starts, it must gain access to the page shared with the in-kernel device driver. To do so, it must first get a port to use to communicate with the driver, using `device_open`. This requires the master device port, which is returned by `task_by_pid(-2)` (but only to the superuser or member of the `kmem` group, which is why this server runs as root). At this point, before the shared page is accessed, the input server sets traps for all Unix signals. These traps will cause some signals to be ignored entirely. Others will cause the device port to be closed, and then the input server will exit. As usual, a thread that does nothing but sleep is detached to receive these signals. Next, `device_map` is called, which will return a memory object suitable for passing to `vm_map`. It is `vm_map` which actually provides the virtual address of the shared page to the user-level process.

This shared page contains the id number of the semaphore that the user-level process must use to wait for an interrupt. When the interrupt handler is called, it reads as many bytes as it can from the MPU-401 into a circular buffer in the shared page, and increments a counter in this shared page that indicates where in the buffer the next byte will be inserted. Once there are no more bytes to be read, the interrupt handler raises the semaphore and returns. The user-level process maintains a second such counter (which is initialized to the value of the counter in the shared page when this page is mapped into user space). When the input server wakes up after waiting for the semaphore, it reads bytes out of the buffer while incrementing its counter until its counter equals the one in the shared page, at which point it waits for the semaphore to be raised again.

After access to the shared page has been gained, the MIDI input server creates a port on which it will listen for IPCs. This port is registered with the netname server with the name ‘‘midi_inputport’’ so clients can easily locate this port. When a client wishes to receive MIDI input, it creates a port of its own and sends it to the MIDI input server. The client requests normal and system-exclusive messages separately, so if it is not interested in one it need not be notified. The data bytes of normal messages are passed inline, because they are at most three bytes long. System exclusive messages are passed as a length and a pointer to a copy-on-write buffer.

As each byte is read out of the shared page, it is passed through a MIDI parser, which reassembles the bytes into complete MIDI messages (adding status bytes where running status was used, moving normal messages out of the middle of system-exclusive messages, etc.). Whenever the parser encounters a complete MIDI message (either normal or system exclusive), all clients interested in this message are notified. The parser timestamps all messages, and the times of these messages are passed along with message contents to the interested clients. (These timestamps are timespecs representing elapsed time since system boot).

The server uses `mach_port_request_notification` to find out when one of its interested client ports is destroyed for any reason; when this happens, it removes this port from its lists of interested clients. This is accomplished by passing the registered server port as the listening port to `mach_port_request_notification`, and passing incoming IPCs through `notify_server`, which is part of `libmach`. This requires that several symbols be defined for the notification port to call back into the MIDI input server, these are:

do_mach_notify_port_deleted,
do_mach_notify_msg_accepted,
do_mach_notify_port_destroyed,
do_mach_notify_no_senders,
do_mach_notify_send_once, and
do_mach_notify_dead_name.

Each of these takes two arguments, both of type mach_port_t, and returns a value of type kern_return_t. Because the only notification ever requested in the MIDI input server is MACH_NOTIFY_DEAD_NAME, all of these functions except do_mach_notify_dead_name do nothing except return KERN_FAILURE (since they should not have been called in the first place). In do_mach_notify_dead_name, the second port passed in is deleted from both (system-exclusive and normal) lists of interested clients if present, and returns KERN_SUCCESS.

4.6. MIDI library input.

When an mi_id is created, the ITC MIDI library automatically creates an input thread which creates a port and registers itself with the input server as interested in both normal and system-exclusive messages. This thread caches the mi_id along with the receiving port that it is associated with in a separate list. When it receives an IPC from the input server, it looks up the associated mi_id from its cache. In this manner, several mi_ids can coexist within the same process. The message is placed in the mi_id's input queue, and a condition indicating that MIDI input has occurred is signalled.

Calls to mi_get (for normal MIDI messages) and mi_getx (for system exclusive messages) examine this input queue, and remove items from it as necessary. We use separate calls because normal MIDI messages have a maximum length of 3 bytes and are easily handled as fixed-length data, whereas system-exclusive messages can have any length. If the queue is empty and the call is specified to block (by a function parameter), the calling thread will be suspended pending the signalling of the condition that MIDI input has occurred.

Like the output threads, the MIDI input thread examines its mi_id when it is awakened to determine if mi_close has been called and the thread should exit.

5. Performance of the MIDI system.

To evaluate the performance of the Mach 3.0 microkernel under real-time constraints, we constructed a simple program which enqueues 100 MIDI messages per second (maximum MIDI bandwidth is about 1000 messages per second), approximately one second in advance. Ideally, the messages would be dispatched one every 10 milliseconds, indicating perfect performance. This proved not to be the case.

Before describing the measurement of MIDI output by Mach drivers, we will describe our instrumentation, consisting of an IBM RS/6000 running AIX 3.1, using a MIDI driver developed earlier. This AIX MIDI driver timestamps incoming messages within the device driver, at high interrupt priority. Before we can make claims about Mach MIDI output drivers, we need to measure the accuracy of the RS/6000 MIDI input drivers. We did this by generating 100 messages per second on another RS/6000 and recording the maximum interarrival times of messages. If both machines were infinitely fast, we would measure exactly 10ms between each incoming message. If however, the receiving machine occasionally takes 1ms to respond to the MIDI receive interrupt, then occasionally the reported interarrival time will be 11ms. In fact, we measured a maximum interarrival time of 10.7ms. This is not necessarily all due to the receiver, but since we are looking for an upper bound, we conservatively use the figure 0.7ms as the worst-case timing error. If we run a file-system intensive process on the receiving RS/6000, we can push the error up to 1.1ms.

Returning to Mach, we examined performance under two conditions. In the first, the output threads are scheduled the same as other threads using a timesharing policy. In the second, fixed priority scheduling is enabled on the processor, and the output threads are set to run at the highest priority. In both cases, the output threads were wired to kernel threads, and the pages containing outgoing messages were wired to prevent the message queues from being swapped.

An important point to note about fixed-priority scheduling is that it is not properly supported by the C-Threads package. C-Threads uses spin locks to avoid costly kernel calls, and it assumes that when a thread yields the processor, that another thread will run. If threads are scheduled according to fixed priorities, a priority-inversion situation may cause starvation. We discovered this only after finishing all of our implementations (based on C-Threads) and measurements. There is no evidence that serious problems occurred, but this problem certainly requires attention. We are implementing our own synchronization primitives for our port to RT Mach.

On an otherwise unloaded system, output intervals fell mostly in the 9.5 to 10.5 millisecond range, with a few variations outside this range. The maximum interval noted with fixed priority threads on an unloaded system was 16.7 milliseconds, indicating that something preempted our threads for about 6.7 milliseconds. Using a timesharing policy on an unloaded system, intervals again fell mostly in the 9.5 to 10.5 millisecond range. The number of variations outside this range was much greater, however, and the maximum observed interval was 21.6 milliseconds, indicating an 11.6ms latency.

Adding a single, compute-bound process to the machine that performed no input or output did not noticeably affect the intervals when fixed priority scheduling was active. Tests using the timesharing policy did not fare as well though; the interval for message arrival times increased to encompass the entire 9.0 to 11.0 millisecond range, and the number of messages falling outside this range increased dramatically. Several delays of more than 200 milliseconds were observed.

Performing a compile in an uncached AFS volume introduced a worst-case scenario. Another process was not only competing for CPU time but was also performing many disk and network accesses. The effect on the timesharing policy client was dramatic; the delays between most messages varied over a range from 6.0 to 14.0 milliseconds, and a large number of messages fell outside this interval. The most dramatic delay between messages was over 10000 milliseconds (ten seconds). The fixed priority client fared much better. Most message delays fell in the 9.0 to 11.0 millisecond range, with a maximum variation of 149.6 milliseconds. This indicates that network threads interfere with real-time performance. The number of delays outside the 9.0 to 11.0 millisecond range was extremely low, and as anticipated, the compile took significantly longer while running with the fixed priority MIDI client.

Figure 1 is a screen dump from our RS/6000 MIDI timing monitor program. The horizontal lines (vertical axis) represent interarrival time in units of 1ms. The interarrival time is plotted on successive horizontal pixels for each incoming MIDI message. When the right edge is reached, the program wraps around to the left edge; thus, the resulting plot is the superposition of several passes from left to right. In a perfect system, all points would fall at the 10ms line. While producing the plot by running a fixed priority task in Mach, we remotely ‘‘`finger`’d’’ a user on the Mach machine. The Mach system’s finger demon fetched a `.plan` file from AFS, resulting in a MIDI delay of more than 60ms. (This is off the scale, but indicated by the ‘‘Worst case’’ number displayed at the lower right corner of the figure.) This illustrates that even a fixed priority program on an ‘‘unloaded machine’’ can fail to meet real-time requirements. Any machine on the internet is capable of making the same `finger` request.

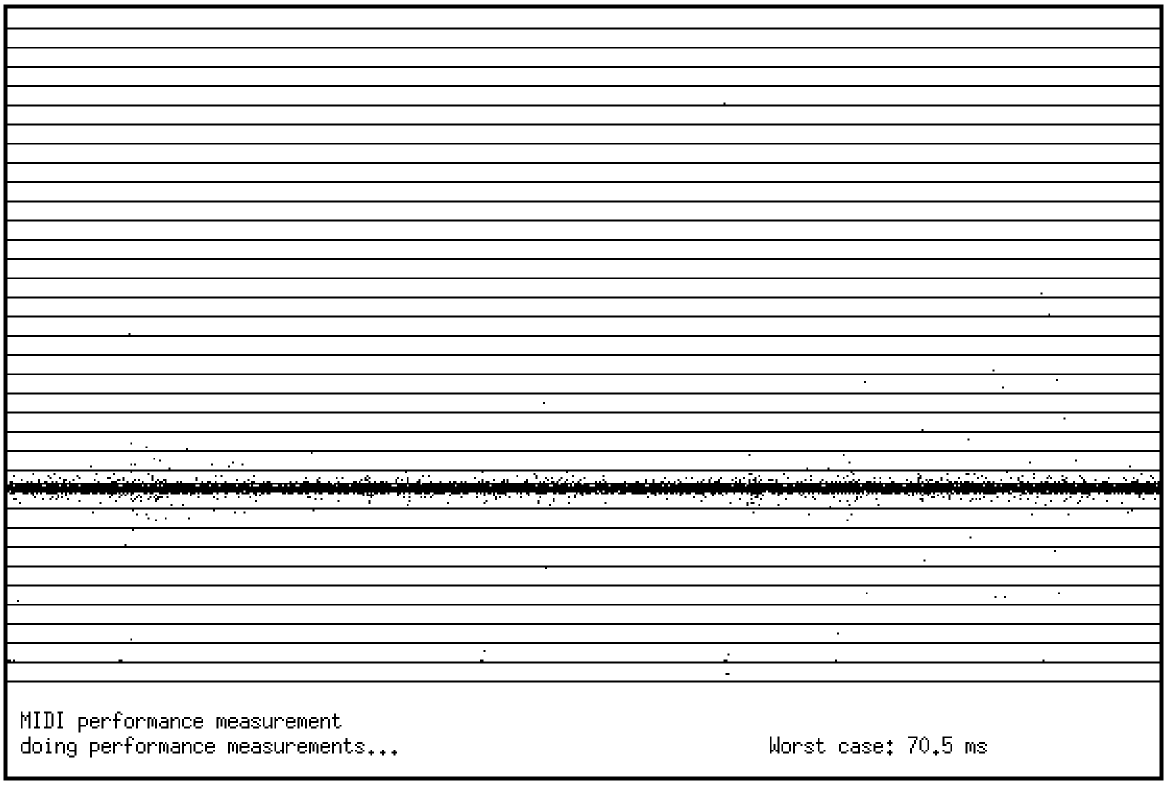

Figure 1: Plot of MIDI message interarrival times. Each line represents 1 millisecond. The data points cluster around the nominal 10 ms line, but notice the scattered points near the right side of the figure indicating longer delays.

For the next test, we ran two xterms on the Mach machine, displaying elsewhere. While the MIDI output clients ran, the mouse pointer of the host where the data was being displayed was passed over the two xterms, without keyboard input or mouse clicking. This generates X11 Enter and Leave notification events (which xterm requests, and uses to redraw its text cursor as either a solid or outlined rectangle). Using the timesharing client, the normal range of message delays was 9.0 to 11.0 millisecond range, with large numbers of messages falling outside this range, distributing themselves over the 1.0 to 18.0 millisecond range. The greatest observed delay was 165.1 milliseconds. When using fixed priority output threads, most messages remained in the normal 9.5 to 10.5 millisecond delay range. Several rapid mouse passes were necessary to trigger a delay in message dispatching, and the largest such delay generated was 21.7 milliseconds.

Keyboard input in an xterm running on and displaying on the Mach machine spread the messages of the timesharing client over a range of 1.0 to 28.0 milliseconds, with the maximum delay of 169.9 milliseconds. Using the fixed-priority client, the normal range of delays was 9.0 to 11.0 milliseconds, with a several messages falling outside this range. The greatest delay was 109.6 milliseconds.

With no load on the system, pressing the return key on the console of the Mach machine had a dramatic effect. Each keypress caused the next message to be dispatched with a 30 to 36 millisecond delay. The greatest observed delay was 36.5 milliseconds. While most such keypresses did not affect the fixed priority output client, about 1 in 10 did cause delays of up to 34.3 milliseconds. This indicates that keyboard input is also a problem.

Mouse input while the Mach machine runs the X window system is another cause of distress. Simply moving the mouse pointer back and forth over a distance of about an inch over the root window, without entering or leaving any windows, caused the normal range of message delays in the timesharing client to fall in the 1.0 to 24.0 millisecond range. The greatest delay was 521.2 milliseconds. The fixed priority client fared somewhat better, most messages fell in the 9.0 to 11.0 millisecond range. Quite a few messages fell outside this range, however, with the greatest observed delay being 20.1 milliseconds. This indicates that mouse handling is a serious problem for real-time applications.

Table 1 summarizes these measurements. In general, fixed priority scheduling helps, but the performance is still intolerable.

Latency	Timesharing	Fixed Priority
No Load	>11ms	>6ms
Computation Load	>200ms	>6ms
AFS Load	>10000ms	>159ms
X11 +Network Load	>155ms	>11ms
X11 Keyboard Input	>159ms	>99ms
Console Keyboard Input	>26ms	>24ms
Local X11 Mouse Input	>511ms	>10ms

Figure 1: Latency table summarizes message delays.

5.1. Summary of interrupts and low-latency scheduling.

While Mach enables programmers to develop device driver code outside the kernel, the term ‘‘user-level’’ is somewhat misleading. In practice a driver consists of an interrupt handler that is compiled into the microkernel, services interrupts, and performs other hardware-level activities. This is not a fundamental problem, as demonstrated by proprietary systems that can dynamically load interrupt handlers, but it is a practical consideration. The bulk of the device driver can run in a user-level process, but the process must have root privileges. Again, this is not a fundamental problem, but a practical consideration.

Performance measurements indicated that there are several issues which need to be addressed for Mach to provide reliable real-time services. While the fixed priority thread scheduling clearly works, there are several high-priority threads in the system which can interfere with real-time processes (such as the keyboard, mouse, and network input threads). In the past, setting these threads to run at the highest possible priority has not interfered with ordinary Unix activities. Unlike ordinary processes, real-time processes cannot be preempted with impunity, even for a few milliseconds. At this time, under the existing Mach, a fixed-priority thread is only assured of running within about 0.15 seconds of its requested run time.

6. Timer overhead measurements.

Another parameter we measured is the overhead of interrupts on the system. This is important because the system clock is typically implemented as a periodic interrupt. If the overhead per interrupt is high, then there is a tradeoff between clock resolution and system throughput. We measured the overhead of the clock by changing its interrupt frequency and measuring the resulting impact on a background process. The background process simply loops to consume all the available processing power. We made sure that the background process read through many lines of code and accessed many memory addresses in an attempt to flush the cache between timer interrupts.

Since the clock interrupt time and the time to perform one iteration of the background loop are both unknown quantities, we made one measurement with an interrupt rate of 100Hz, which allowed 1530 iterations per second, and another with an interrupt rate of 2000Hz, which allowed 1333.6 iterations per second. Solving for clock interrupt time, we get 67us. There is some variation from one run to the next, even running on a stand-alone system, but this measurement is repeatable to within a percent or two.

Given the clock interrupt time, we can estimate that a 1KHz clock rate will consume 6.7% of the available processing time. This seems to indicate there is a trade-off between clock resolution and processing overhead: the better the resolution, the more clock interrupts there are, and the more processing time is wasted. It appears to be possible, however, to use the time-of-day clock on PC/AT hardware to generate low-frequency periodic interrupts, freeing the currently used timer for high-resolution interval timing. This would reduce overhead because the interval timer could be set to interrupt only at times when processing is required. At the same time, resolution would improve because the timer resolution is a fraction of a microsecond.

7. Digital video applications.

Digital video typically requires high-bandwidth data transfer across several process boundaries, so it is an interesting test for operating systems. We implemented a digital video playback system as part of our multimedia testbed. We will describe the application and then present measurements of various components of the system.

7.1. Tactus.

The digital video application is really a general purpose multimedia playback system. The system is based on Tactus, a multimedia extension for the Andrew Toolkit. Since Tactus has been described elsewhere [Dannenberg 93], we will only summarize the basic mechanisms of

Tactus here. Tactus is based on the principle that if multimedia data can be precomputed or prefetched into buffers, then the data can be delivered to devices with low latency and accurate timing. (See Figure 2.) A problem with this idea is that when data is prefetched and buffered, it can be difficult to maintain synchronization between streams. Therefore, Tactus expects data to be timestamped. Tactus uses timestamps to achieve and maintain synchronization.

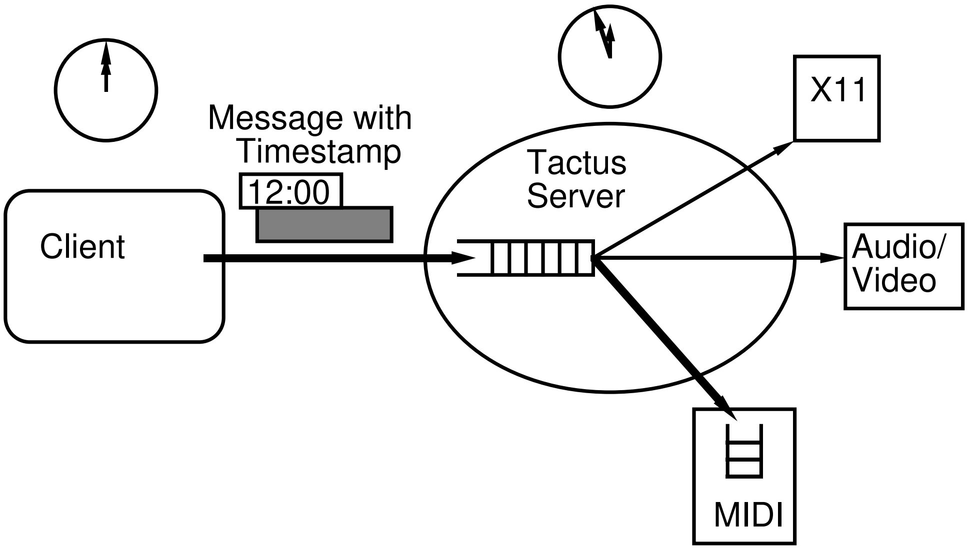

Figure 2: The Tactus System. Clients send timestamped data (heavy lines) to the server ahead of real time. Data is buffered and then delivered to various presentations devices. Some presentation devices (e.g. MIDI as shown here) may accept data early and provide further buffering and more accurate timing than can be provided by the Tactus Server. The clock on the left shows logical time as seen by the client, while the clock on the right shows real time as seen by the Tactus Server.

Tactus is implemented in two parts. The Tactus server is a process that buffers and synchronizes timestamped streams of multimedia data. The Tactus toolkit extensions form a library that helps programmers to schedule data-producing computations. The toolkit also generates timestamps for the data and delivers the data to Tactus. As a result, the programmer can often write multimedia programs as a collection of simple ‘‘active’’ objects, which request to be run at specific times. Tactus automatically runs the objects early, timestamps the output, delivers it to the server, and synchronizes the output with streams from other objects. This greatly simplifies the programming of a real-time, concurrent system.

7.2. The Application.

Our multimedia playback system will now be described. The user selects a presentation by clicking on an icon in a scrolling window. The application then begins reading multimedia data from disk and sending it to the Tactus synchronization server. The Tactus server forwards data to various devices and maintains synchronization between multiple data streams. The Tactus server runs at a higher priority than the application, and time-critical threads in Tactus avoid making calls to the Unix server, so Tactus has tighter real-time properties than its clients.

In order for this system to work, we need efficient data transfer through a number of processes: Data moves from disk to application to Tactus to the X11 display server. Data movement requires memory allocation, memory copy, and release as well as synchronization between processes. We explored various implementations and report on their performance below.

8. Memory Transfer.

There are many ways to transfer data. We begin by timing repeated calls to `bcopy`, copying from one memory buffer to another. Copies of multiples of 4 bytes ran at 10MB/s to 14MB/s, based on 10,000 to 100,000 iterations and blocksizes ranging from 656 to 65536 bytes. We have not yet explored the observed variance. When blocksizes are not multiples of 4 bytes, the transfer rate is between 3.5MB/s and 4.0MB/s.

For comparison, an RS/6000 model 530 copies about 45MB/s, a factor of 3 to 4 improvement.

8.1. In-line IPC.

Mach IPC messages can transfer data *in-line*, where data is copied, or *out-of-line*, where the memory is mapped but not copied. In-line messages of various sizes were sent from a ‘‘client’’ to a ‘‘server’’ process; the client blocked waiting for a reply, and the server always replied by simply returning `KERN_SUCCESS`. For each message size, 1000 iterations were performed. As expected, times increased with buffer size. A 1-byte buffer took 0.160ms, while a 32768-byte buffer took 8.77ms. For N bytes, the transfer time is approximately (0.160 + N/4000) ms. In other words, each message incurs a 0.160ms overhead, and data is transferred at about 4MB/s. Note that this time includes the application overhead imposed by MIG, the Mach Interface Generator.

8.2. Out-of-line IPC.

Out-of-line IPC messages were measured in a similar test. In this case, the receiver performed a `vm_deallocate` to remove the data from its virtual address space. Without this call, virtual memory becomes cluttered with messages, and the time to send a message was observed to increase linearly with the number of messages sent. Since no data is actually copied, the time to send any number of bytes is approximately constant, but there is a linear cost of mapping pages if the data is examined.

We measured between 0.36 and 0.39ms simply to send a message. In this case, the sender and receiver do not actually touch the data. Instead, the receiver simply deallocates the out-of-line data and sends an in-line acknowledgment message back to the sender. We subtracted 0.16ms from the measured round trip time to obtain the one-way out-of-line IPC message time

When the overhead of accessing pages is added, the real transfer time is approximately (0.6 + N/6300)ms. For this measurement, the sender copied data into the buffer to be sent and the receiver touched the first byte of each page, forcing the page to be mapped. The data was then deallocated and a reply message was sent. Starting with the measured round trip time, we estimated and subtracted the bcopy time and the reply message time. We then calculated the per page cost by timing messages from 4K to 32K in length.

A more realistic test is to have the sender allocate new data for each message, and to deallocate the data upon sending it. To understand why this is so, consider how out-of-line messages might be used for a multimedia data stream. The sender wants to send a sequence of messages to the receiver, but the receiver does not want to deallocate the received data until it has been used. When a message is sent from the sender, it is marked by Mach as ‘‘copy on write,’’ meaning that a write to the buffer will cause a page fault (otherwise the write would destroy the data previously sent and waiting for use by the receiver.) The solution is for the sender to allocate new data for each message, avoiding the needless copy on write. In practice, the sender will still generate page faults when the newly allocated memory is touched, and these will require zero-fill, but this should be slightly cheaper than a copy.

Allocation of each message by the server increases the message send time to between 0.46 and 0.54ms/message. Neither the sender nor receiver actually touches the data in this case. We subtracted 0.16ms, representing the reply message, from the measured round trip time to estimate the one-way out-of-line IPC message time.

Even when memory is ‘‘allocated,’’ no physical memory is actually associated with the address until memory is read or written. We modified the test so that the sender uses bcopy to fill the message before sending it, and the receiver touches the first byte of each page. The transfer time is approximately (1.1 + N/8500)ms.

8.3. Socket I/O.

Unix sockets are convenient, but socket I/O involves the Unix server. Thus, the overhead of using sockets is quite high. For our measurement, we had a client send data to a server, which then sent the same data back to the client. Because the client and server are symmetric, we can divide the total time by 2 to get one-way socket transfer times. The one-way time for a one-byte transfer is about 2.9ms. The transfer time grows somewhat non-linearly with message size, and 4096 byte messages provide faster throughput (at about 600KB/s) than 16384 byte messages (at about 540KB/s). This may be the effect of the secondary cache. Note that these times are for Mach 3.0. Sockets should be faster under Mach 2.5 because there, sockets are implemented in the kernel. Also note that optimization of this code is ongoing, and substantially better performance may be seen in future Mach releases.

8.4. Shared Memory IPC.

Rather than using Mach IPC, we implemented our own communication mechanism using shared memory to eliminate data copies, and simple Mach IPC messages for synchronization and notification. The timing test consists of a client allocating a buffer from shared memory, copying data to the buffer, and notifying the server via IPC. The server performs symmetrically, allocating a reply message, copying data from the incoming message to the reply, and notifying the client. As before, one-way message transfer times can be obtained by dividing the total time by 2. Unlike the previous tests, this test used the actual Tactus server which uses multiple threads to receive messages and which adds headers to messages. The one-way message time is 4.4ms. We have not yet measured where all of this time is going.

A fair comparison with the previous methods would be a streamlined implementation where the client and server were single threads that signaled each other with simple IPC messages. This is essentially equivalent to the in-line IPC case where only a few bytes are transmitted. The measured time was 0.16ms. This assumes that no data must be copied into the shared memory area. For example, in our application, we read images directly from the disk into the shared memory for transmission to the Tactus server.

8.5. Summary.

Figure 3 illustrates memory copy strategies and their performance. The fastest transfers by far were seen with a shared memory interface, where data is neither mapped nor copied. Unix sockets are very slow by comparison. In-line IPC offers an intermediate rate. At first, we thought out-of-line IPC would be ideal for mapping large amounts of data without actually copying it. Several factors make this not the case. First, newly allocated memory must be zeroed; this causes a time-consuming page fault as well as a zero-fill operation. This alone is about as expensive as a memory copy. To avoid allocating new memory, old memory must be retained. This is possible, but it means that neither the sender nor the receiver can write into the data or else a copy-on-write fault will be taken. It seems that the optimal implementation is to ‘‘recycle’’ memory: The client starts with a pool of messages which are deallocated upon sending to the server. After the server finishes using a message, the message is returned to the client, this time deallocating the message from the server’s memory. Since out-of-line IPC can move data from anywhere in an address space, this scheme might be advantageous if large messages needed to be routed to multiple destinations. Otherwise, the shared memory interface is simpler and faster.

9. File I/O.

Our combined video and audio data rate is about 220KB/s, which is a sizable fraction of the total disk I/O bandwidth on our machines. We measured how much time was involved in reading the disks. In these tests, the IDE drive is a Western Digital 300 MB WDAP4200 with a Gateway MIO400 controller. The SCSI drive is a Seagate 1GB drive with an Adaptec 1542b controller.

Using `stdio` and IDE drives, the transfer rate is about 440 KB/s. Fortunately, disk I/O does not require 100% of the CPU. Depending upon the method used to read data, the CPU utilization ranges from 31% to 51% when reading as fast as possible. This means that while we are using about half the maximum disk bandwidth, we are only using about one fifth of the CPU for disk reads.

We experimented with various ways to read files. These methods are summarized below.

- *stdio*: using the Unix `stdio` library.
- *block*: using Unix `read` commands directly.

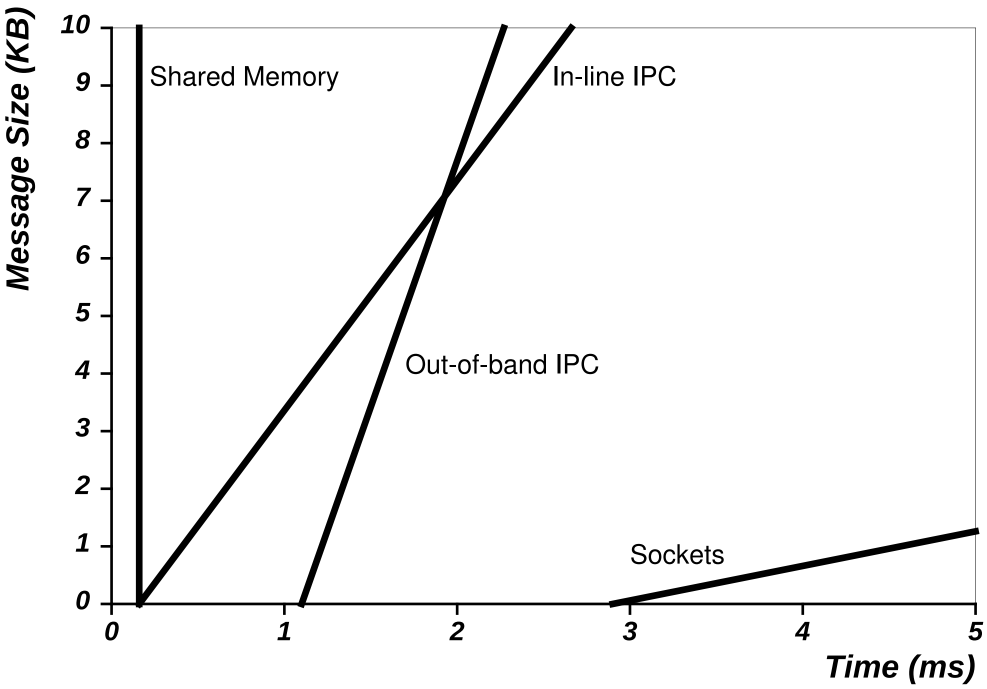

Figure 3: Performance of various forms of communication in Mach.

- *memory-mapped*: using Mach mechanism to map files regions into user address space and faulting in pages.
- *mapio*: a partial reimplementation of the Unix `stdio` library interface using the memory mapped files capability of Mach.

The method and the maximum performance for SCSI and IDE drives are presented in Table 2. The block sizes in the table refer to the size of the read requests issued by the applications. Many block sizes were tried, but only the size that gave the best performance is reported here.

A few words are in order about these measurements. Within a column (SCSI or IDE) all numbers represent reading the same file and should be comparable. The relative performance of SCSI vs. IDE, however, is not as meaningful because only one file was used, and the layout of the file on the disk was not controlled. To avoid caching effects, a 30MB file was used, and the main memory size is 16MB. All of these tests used a 33Mhz i486 processor. These tests were done in single user mode and were consistently repeatable, typically to within 2% and always within about 5%.

The numbers show that block I/O provides the fastest access. This is not consistently the case, and under some conditions mapped I/O is faster than block I/O (see below). The last line of the table gives a performance figure for DOS running on identical hardware. Although this figure is high relative to Mach, it was obtained with a different file, and the timing difference may be due to the physical layout of blocks on the disk. Overall, performance seems to be dominated by (lack of) tuning, and not by the access method.

Our measurements of CPU time include the application, the kernel, and the Unix server. CPU time is computed by subtracting idle time, as measured by the program `freetime`, from the

Method	SCSI KB/s (blocksize) CPU%	IDE KB/s (blocksize) CPU%
stdio	840 (1KB) 102%	450 (64KB) 51%
block	940 (8KB) 112%	440 (8KB) 42%
memory-mapped	660 (1MB) 54%	420 (256KB) 31%
mapio	660 (20KB) 68%	430 (2 MB) 38%
DOS	-	850 (32KB)

Figure 2: Disk read performance using different input methods.

total run time. The `freetime` program uses a low-priority task to consume and count all available idle CPU cycles while other programs are running. Unfortunately, this interacts with disk reading programs. We conjecture that critical disk timing is perturbed by the addition of even low-priority processes. In one extreme case, run time consistently jumps from 32 to 47s when `freetime` is run.

To calculate a percentage of CPU utilization, we took the ratio of measured CPU time (with `freetime`) to actual run time (without `freetime`). This sometimes results in an inflated value that exceeds 100%. In spite of these anomalies, these timing numbers are reproducible.

Other anomalies were encountered. Mapped I/O is much faster when running in multi-user than in single-user modes. For example, on a 66Mhz 486DX system, mapped I/O run time dropped from 48 to 30s when multiuser mode was enabled. On the other hand, block I/O and stdio run times stayed roughly constant.

We re-ran all of the timing tests on a 66Mhz 486DX system for comparison. The 486DX system has a faster CPU clock but the same 33MHz bus as our standard system. With the fast CPU, we found relatively little improvement in I/O. This could be explained by saying the system is I/O bound, but if this were the case, we would expect the CPU utilization to go down with a faster processor. This did not occur, indicating that a faster CPU clock speed alone does not lead to dramatic improvements.

Overall, we found performance to be rather erratic. A variation of 20% between two runs in multiuser mode is not uncommon, and much higher variations, up to 70%, were observed. In cases where there is large variation, we see correspondingly large amounts of paging activity, although we cannot explain this paging in terms of our application. We have also seen performance vary by 50% or more when running the `freetime` program concurrently with our I/O test program. It seems that the Mach strategies for caching pages, disk scheduling, and process scheduling make the system very unpredictable. Consequently, the reported I/O bandwidth may be as much as double the bandwidth usable in practice.

The erratic performance of disk I/O may be a result of overall unpredictability in the kernel. If so, then better real-time support could lead to higher system performance overall.

10. X11 Performance.

Our application creates video by displaying standard X11 images at 10 frames/s, thus X11 performance is a critical component. The standard socket-based X11 performance was a problem, so we used a shared-memory interface [Ginsberg 93] to speed up data transfers.

With a standard socket-based X11, we wrote a program to repeatedly call `XPutImage`, `XFlush`, and `XSync`, and we measured the time per frame. The display time ranged from 76 to 85ms for a 19.2KB image; in other words, we can display about 240KB/s.

With our shared-memory X11, the display time ranged from 10 to 23ms per image, for a transfer rate of from 835KB/s to 1.9MB/s. We did not track down the source of this variance, but it is obvious that this shared memory interface is significantly faster. We created a special libX11.a that has a modified version of `select` so that ordinary X clients need only to be relinked to use the new interface.

11. Overall Timing Model.

Combining our measurements, we can build a model of where time is spent in our video mail application. We found it convenient to work in terms of computation time per frame. Our current application assumes 10 frames per second, so the total budget is 100ms per frame. The operations and their times are as follows:

Disk read	20ms
Send data (client to Tactus)	9ms
Copy data within Tactus	2ms
Send and display data (Tactus to X11)	20ms
Total	**51ms**

This model does not include 15KB/s audio, which follows roughly the same data path. We have not completed measurements, but we expect audio to add another 10% to the total processing time per frame. Although these numbers indicate a fair amount of ‘‘headroom,’’ this application barely keeps up with real time. This is disturbing, but consistent with the large performance variations we have observed in our measurements.

If we were to use Unix sockets throughout, the total processing time for video alone would be about 170ms, much slower than our shared memory implementation, and slower than real time.

12. Conclusions and Future Work.

Our low-latency measurements show that Mach is far from supporting any kind of reliable low-latency interaction, even with fixed-priority scheduling. This seems to be a combination of scheduler problems (the scheduler will allow a low-priority thread to run for 10ms before preempting it), and server problems (various high-computation device drivers and servers ‘‘take over’’ the processor for extended periods of time at high priority, in contradiction to well-known scheduling principles). Mach emphasizes a lazy evaluation strategy and was not designed as a real-time system, so it should not be too surprising that the worst-case latency is fairly high.

Our measurements show that shared memory interfaces support high-bandwidth communication of multimedia data. Shared memory eliminates expensive page mapping and our application and server only perform one data copy (from the client-Tactus interface to the Tactus-X11 interface). With a little work, we could eliminate this copy as well. Further improvements might be made within the Unix file system and within X11.

Many of our measurements are preliminary, and exhibit a fairly wide variance. Running a machine standalone seems to eliminate all but a percent or two of performance variation. This is somewhat disturbing because it implies the overhead of maintaining connections to a network and file systems is quite high, unpredictable, and only marginally related to the currently running tasks. This in turn has serious implications for providing real-time services and predictable performance to users.

Similar comments apply to file system performance. Caching and scheduling in Mach apparently conspire against predictable performance. The worst-case performance often approaches a factor of 2 worse than ‘‘normal’’ performance. For real-time and multimedia systems that are limited by the worst-case performance, Mach imposes a very high overhead.

One particular result we would like is a more accurate model for interprocess communication via shared memory, in-line IPC, and out-of-line IPC. For example, we would expect in-line IPC transfer rates to be some integer factor slower than `bcopy`, based on the model that IPC has a fixed overhead plus some integer number of memory copies.

Our measurements and experience indicate that shared memory interfaces can provide significant performance improvements over those based on mapping or copying, although shared memory interfaces do sacrifice some amount of protection. Message-based interface *abstractions* can be based on shared memory, and could be supported by operating systems and interface generators.

In general, root privileges are required too often. For example, the shared-memory X11 server can crash the machine, as can tasks that install user-level device drivers. The Mach netmemory server is a safer way to obtain shared memory (without root privileges). Better still would be a general shared memory IPC facility. Are there other possibilities for realizing better protection without losing efficiency?

We began this work with the idea that applications could run directly on the Mach microkernel, avoiding the overhead and unpredictable behavior associated with the Unix server. In practice, we found it very difficult to avoid the Unix server. The file system must be accessed via the server, and most of our programs have extra threads to `sleep` waiting for signals and to `select` waiting for input as well. Even when direct Unix calls are avoided, the paging, interrupt, cache, network, and disk behavior of the server have a major impact on the performance of other tasks. One of the advantages of the user-level device driver implementation is that devices are mainly controlled by ordinary threads that can be scheduled in competition with application threads [Forin 91]. This should reduce the amount by which devices interfere with time-critical processes. It remains to be seen how successfully RT Mach will isolate applications from interaction with the Unix server.

RT Mach extends Mach 3.0 with real-time threads and scheduling which should greatly enhance our low-latency applications. RT Mach should even improve upon the performance of Tactus, allowing the Tactus server and X11 to run at high priority and low latency without interference from the application and file system, which have less stringent latency requirements. We have already begun to port our applications to RT Mach.

A limitation of our work is that we have not studied the architectural dependencies of our results. Mach 3.0 runs on other machine types, and it would be interesting to perform similar measurements on these machines.

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 93] R. B. Dannenberg, T. Neuendorffer, J. M. Newcomer, D. Rubine, D. Anderson. Tactus: Toolkit-level Support for Synchronized Interactive Multimedia. *Multimedia Systems Journal* , 1993. (to appear).

[Forin 91] A. Forin, D. Golub, B. Bershad. An I/O System for Mach 3.0. In *Proceedings of the Second USENIX Mach Symposium*, pages 163-176. USENIX Association, 1991. Also available via FTP from [mach.cs.cmu.edu] /usr/mach/public/doc/published/IO.ps.

[Ginsberg 93] M. Ginsberg, R. V. Baron, and B. N. Bershad. Using the Mach Communication Primitives in X11. In *Proceedings of the Third USENIX Mach Symposium*, pages 103-110. USENIX Association, 1993.

[Golub 93] D. B. Golub, G. G. Sotomayor, Jr., F. L. Rawson III. An Architecture for Device Drivers Executing as User-Level Tasks. In *Proceedings of the Third USENIX Mach Symposium*, pages 153-171. USENIX Association, 1993.

[Nakajima 91] J. Nakajima, 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] J. Nakajima, 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.

[Tokuda 90] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Toward a Predictable Real-Time System. In *Proceedings of the USENIX Mach Workshop*. USENIX, October, 1990.

Table of Contents

List of Figures

List of Tables