RT-IPC: An IPC Extension for Real-Time Mach

Takuro Kitayama[1]*, Tatsuo Nakajima*[2]*, and Hideyuki Tokuda*[1]

[1] *School of Computer Science, Carnegie Mellon University*
{takuro, hxt}@cs.cmu.edu
[2] *Japan Advanced Institute of Science and Technology*
tatsuo@jaist-east.ac.jp

Abstract

Interprocess communication (IPC) provides the fundamental mechanism for the Mach microkernel to be extensible and flexible. Mach IPC provides efficient communication mechanisms for many applications. However, it does not provide sufficient functionality for real-time applications which have rigid timing constraints among threads. In Real-Time Mach (RT-Mach), we have extended Mach IPC to be *priority inversion free* for real-time applications.

This paper describes the Real-Time IPC (RT-IPC) facilities, its implementation, and the evaluation results. We used the Distributed Hartstone (DHS) real-time benchmark for the evaluation and the results show that RT-IPC can reduce priority inversion and improve CPU utilization for real-time applications.

1. Introduction

In traditional operating systems, it is difficult to support real-time applications such as continuous media applications and mobile computing, due to the lack of facilities which can manage time constrained resources, such as processor cycles, memory, communication ports, shared variables, and so on. To support real-time applications, we need time-driven resource management for all such resources.

Real-Time Mach (RT-Mach) has been developed at CMU to provide a common distributed real-time computing environment[19]. In RT-Mach, we have developed a real-time thread model where a periodic activity can be easily defined by explicitly specifying a timing attribute. It also provides real-time scheduling services, real-time synchronization[18], and high resolution clocks and timers[14].

The original Mach microkernel provides a port-based IPC mechanism. However, it is designed and optimized to provide fair services to all type of application programs[2, 3]. In the timesharing environment, it is very rare that two or more messages are queued in one message queue, usually, messages are delivered from a sender to a receiver without queueing, i.e., the average queue length would be less than one.

However, in the real-time environment, several messages can be queued in one queue and wait to be serviced by the receiver. For example, if a video playing server is not fast enough to process several requests within a certain time, some requests must be queued. Furthermore, those messages

This research was supported in part by the U.S. NRaD under contract number N66001-87-C-0155, by the Office of Naval Research under contract number N00014-84-K-0734, by the Defense Advanced Research Projects Agency, ARPA Order No. 7330 under contract number MDA72-90-C-0035, and by the Federal Systems Division of IBM Corporation under University Agreement YA-278067. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of NRaD, ONR, DARPA, IBM, Northrop, Bellcore, or the U.S. Government.

should have a priority according to user's interest. Data which will be displayed on a background window can be delayed, but data which is currently in an active window should be displayed within a certain time. We need a mechanism which can guarantee message delivery in a timely fashion.

For real-time synchronization, there has been developed many mechanisms to avoid unbounded priority inversion problems[12, 15]. However, the most of the real-time synchronizer are designed without integrated with real-time IPC. In general, the priority inversion problem in client-server communication is more serious one, since the length of priority inversion tends to be much longer than that of synchronization. In this paper, we propose a Real-Time IPC (RT-IPC) mechanism to support real-time applications, describe its implementation in RT-Mach, and show the performance evaluation results of RT-IPC. In Section 2, we show the overview of RT-Mach and the original Mach IPC, then describe the problems with the conventional IPC mechanism. Section 3 discusses our RT-IPC model. Section 4 shows the implementation of RT-IPC in RT-Mach. Section 5 shows the performance evaluation of RT-IPC. Related work is discussed in Section 6, and we conclude in Section 7.

2. Motivation

In this section, we summarize Real-Time Mach, and the original Mach IPC, then we describe the problem with the conventional IPC mechanism.

2.1. Overview of Real-Time Mach

The objective of Real-Time Mach is to develop a real-time version of Mach which supports a predictable real-time computing environment. The major features of Real-Time Mach are summarized as follows:

Real-Time Thread Model : A thread is defined for a real-time or non-real-time activity. For a real-time thread, additional timing attributes must be defined by a *thread attribute*. A real-time thread can be also defined as a *periodic* or *aperiodic* thread based on the nature of its activity.

Real-Time Scheduler : In real-time operating systems, thread scheduling plays an important role in managing the system resources in a timely fashion. The RT-Mach scheduler provides us a good flexibility by supporting multiple scheduling policy. Currently, The RT-Mach scheduler supports Time-Sharing, Fixed Priority, Rate Monotonic, Deadline Monotonic, and Earliest Deadline First. A scheduling policy can be set as one of *processor set attributes*.

High Resolution Clock and Timer : Clocks are devices which measure the passage of time and support the use of timers to a particular degree of accuracy. A clock device can be used for accurate timestamps, measurements of CPU usage, or for basing representations of the time of day. Timers are active objects which allow users to synchronize with time in a variety of ways. Timers are kernel-exported objects with three properties; an expiration time, a synchronization action to be taken, and an activity state.

Real-Time Synchronization : Traditional synchronization primitives use FIFO ordering among threads waiting to enter a critical section. In real-time environment, however, FIFO ordering often creates a priority inversion problem where a low priority thread blocks a high priority thread. RT-Mach supports various synchronization policies to avoid unbounded priority inversion.

2.2. Overview of Mach IPC

IPC plays a very important role in the Mach microkernel to provide communication between tasks, especially clients and servers. Since the Mach microkernel provides limited services of its own, a task needs to communicate with many other tasks which provide a variety of services such as file systems and network services. These communication channels are called *ports*. A port is a unidirectional channel consisting of queue that holds *messages*. A message is a typed collection of data. A port is named by *port rights* held by tasks. A task has a *port name space* which collects port names. A task can manipulate a port only if it holds appropriate port rights. Only one task can hold the *receive right* for a port. This one task is allowed to receive message from the port. Multiple tasks can hold *send right* to the port that allow them to send messages into the queue. A task communicates with another task by building a data structure that contains a set of typed data elements[1], and then performing a message-send operation on a port for which it holds send rights. The message is queued onto the message queue in FIFO order. At some later time, the task with receive rights to that port will perform a message-receive operation. The message is logically copied into the receiving task. Multiple threads within the receiving task can receive messages from a given port, but only one thread will receive any given message.

2.3. Basic Issues

IPC is also heavily used in the RT-Mach environment to provide a variety of services by server programs. The original Mach IPC facilities provide fair services for timesharing applications. However, in the real-time environment, this causes unbounded priority inversion problems. In order to build servers for providing various real-time services on top of RT-Mach, we must have a real-time version of IPC.

There are four categories where problems occur with the original Mach IPC: message queueing order, priority hand-off, priority inheritance, and message distribution. In this section, we discuss the four issues.

2.3.1. Message Queueing Order

For timesharing systems, a FIFO queuing policy is acceptable because it ensures fairness. In the real-time environment, however, this may cause the unbounded priority inversion problem where a higher priority message is sent when lower priority messages are on the queue. The processing of the higher priority message has to wait until all processing of lower priority messages are completed. Thus, we need priority-based queueing for real-time applications.

2.3.2. Priority Hand-off

Since the sender and the receiver threads are independent scheduling entities, there should be a mechanism to maintain a priority of receiver when the receiver receives a message. If the priority of the server is lower than that of the sender or message, the processing for the message is interrupted by any other medium priority thread. If the priority of the server is higher than that of the sender, then a low priority request may block any other high priority activities. The receiver thread must propagate the priority from the sender or give a specified priority.

2.3.3. Priority Inheritance

The other issue arises when a high priority message is sent while the server is processing for a lower priority message. The high priority message must be queued on the queue and wait for the completion of the lower priority message processing. At this time, if a medium priority thread becomes runnable, then the receiver thread is preempted by the medium priority thread, even though the higher priority thread is waiting for service. The receiver thread has to inherit the priority from the higher priority sender when the higher priority thread sends a message. This solution is called a priority inheritance protocol[16] and has been developed and applied for real-time synchronization. The basic idea is that a low priority thread inherits the priority of a high priority thread when a high priority thread is delayed by the low priority thread. We need the same mechanism for the real-time IPC. When a high priority thread sends a message, then the priority of the sender should be inherited to the receiver thread.

[1] The user is not required to generate there. MIG (Mach Interface Generator) generates the necessary stub routines.

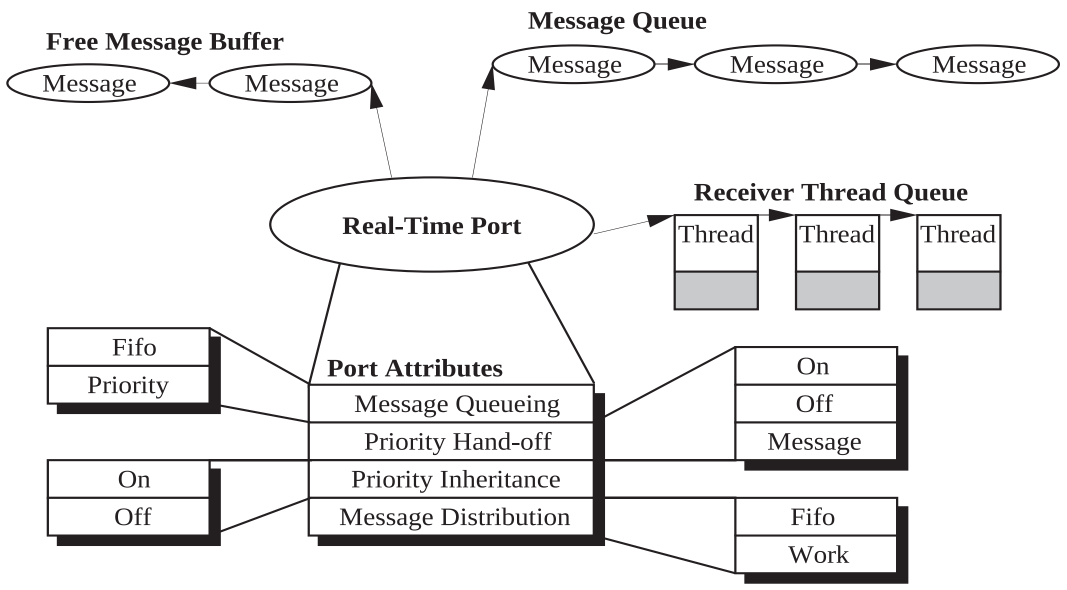

Figure 1: Real-Time IPC Model

2.3.4. Message Distribution

In case there are two or more receivers in a server, there should be a mechanism to determine which receiver should process the message. If there is a sufficient number of receivers running, we do not need to worry about the situation. However, if there are insufficient, we need the mechanisms to select which receiver thread will process the message and bump up its priority if the priority inheritance protocol is activated.

3. Real-Time IPC Model

In order to support real-time applications, we propose the following RT-IPC model. Mach IPC guarantees that messages are received in the order in which they were sent, but RT-IPC could not guarantee that order, since it may cause priority inversions. Our goal is to provide a predictable IPC mechanism for real-time applications.

Figure 1 depicts the basic model of our RT-IPC. We discuss the model focusing on the difference from the original Mach IPC model.

3.1. Port and Receiver Association

RT-IPC supports two kinds of transmission modes. One is synchronous transmission in which the sender is blocked by the receiver until it receives the reply message. The other is asynchronous transmission in which the sender is never blocked by the receiver process, i.e., sender does not wait for the reply message from the receiver. In the both modes, a receiver thread must declare itself to act as the receiver of a port before initiating a communication. By this declaration, the server thread is associated with the port, and its priority will be maintained according to the selected port attribute which is described below.

3.2. Message Buffer

In RT-IPC, message buffers should be allocated before communication to eliminate unpredictable buffer allocation delay. Users need to determine the actual size and number of buffers before initiating the communication.

3.3. Port Attributes

Related to the queueing and priority manipulation, we define four attributes for the real-time port and provide various policies for each attributes. These attributes are summarized as follows.

Message Queueing : It maintains the message queue ordering. FIFO and priority-based ordering policies can be used.

Priority Hand-off : Priority Hand-off manipulates the receiver's priority when a message is transferred. When it is disabled, the priority of the receiver is not changed. When it is enabled, the priority of the receiver is propagated from that of the sender or given priority according to the selected policy.

Priority Inheritance : It executes the priority inheritance protocol[16]. If it is activated, the server inherits the priority of the sender thread which sent the highest priority message.

Message Distribution : It selects a proper receiver thread when two or more receivers are running. Arbitrary or priority-based(WORK) selection can be specified as a policy. When the arbitrary policy is specified, a receiver thread is chosen in FIFO order. The priority-based policy selects a receiver thread according to a given priority.

3.4. Integration with Synchronization

Many mechanisms have been developed for real-time synchronization to avoid unbounded priority inversions. However, synchronization mechanism alone does not avoid the problems, especially in microkernel environments. Application program could be decomposed into several tasks, and they need to interact and communicate each other. A message may be sent in a critical region, a receiver thread has a critical region, or a sender thread can be also a receiver thread of another message.

In order to avoid unbounded priority inversion between synchronization and IPC, we need to maintain priorities consistently among threads and among all critical region and IPC calls.

For example, there are three threads T_H, T_M, and T_L, where T_H has the highest priority, T_M is medium, and T_L has the lowest priority, respectively. When T_M is in a critical section, T_H is trying to get into the critical section, and T_M sends a message to T_L, then priority hand-off happens in different way between the two transmission modes. In synchronous transmission mode, priority of T_H is propagated to T_L, since T_H has to wait the completion of T_L to go into the critical section. In asynchronous transmission mode, the native priority of T_M is passed to T_L, because T_H will never be block by the computation of T_L.

A similar situation may happen if IPCs are called in a nested fashion. If priority inheritance occurs, the priority of all message receiver threads in the nested call will be affected in synchronous mode. In asynchronous mode, on the other hand, priority inheritance affects only the first receiver thread.

4. Implementation

We have implemented RT-IPC in the RT-Mach microkernel to support the queueing and priority management, which were described in the previous section, to provide a better communication mechanism for real-time applications. In this section, we describe the implementation and interface of RT-IPC.

4.1. Implementation Strategy

The interface must follow the original Mach IPC and the rest of the Real-Time Mach Interface such as Real-Time thread and synchronization to provide the uniform interface for programmers.

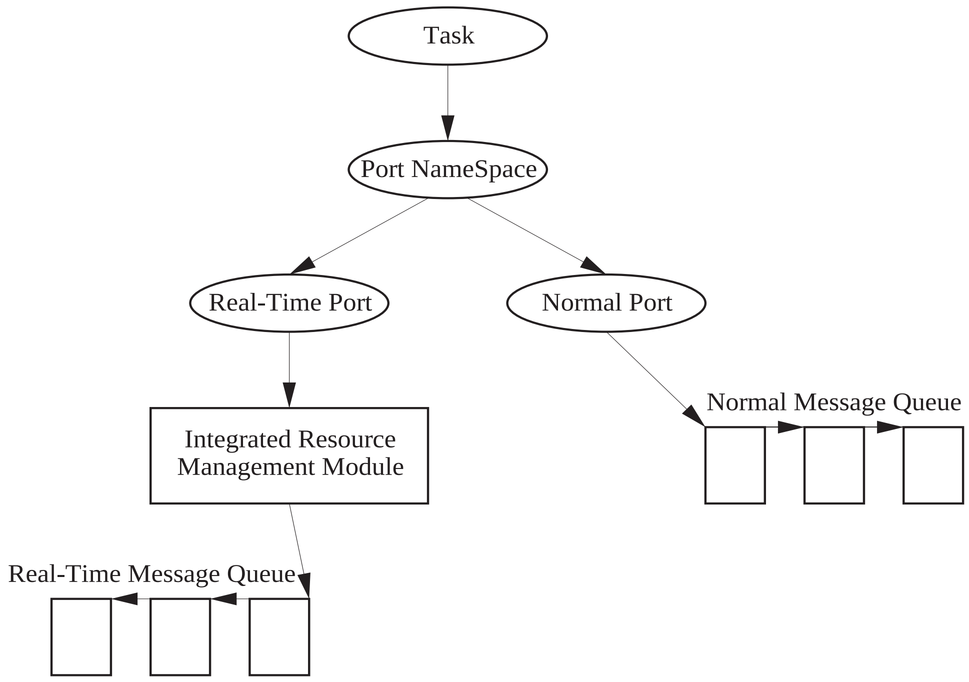

Figure 2: IPC and Real-Time IPC Structure

Figure 2 illustrates the coexistence of the normal IPC and RT-IPC. Each task has one port name space which is sheared by normal IPC and RT-IPC. However, the structure of normal ports and rt-ports are different. A *real-time port(rt-port)* is the real-time version of communication channel(port). A normal port has a message queue to hold waiting messages. A RT-port has some function pointers to the Integrated Resource Management module, which is describe below, instead of having a message queue directly. Each thread can send and receive both normal and real-time messages from and to a normal port or RT-port, if the thread has appropriate right for the ports. Normal ports can handle only normal messages, and RT-ports can handle only real-time messages.

Another main difference is kernel message buffer management. In the original Mach IPC, the microkernel may have one kernel buffer allocated for small size of messages. In RT-IPC, however, all kernel messages are allocated at the port allocation time.

We also consider the effect on the original Mach IPC, it must be minimal. The interface of original Mach IPC has not been changed, we just added some code in the microkernel to check whether the port is a real-time or normal port.

4.2. Basic Structure of RT-IPC

The basic structure of the RT-IPC module in the microkernel is shown in Figure 3. It is roughly divided into two parts. One is the real-time port management module, the other is the Integrated Resource Management (IRM) module. The real-time port management module manages ports, port rights, port name spaces, port sets, and messages in the same fashion as the original Mach IPC[6, 2].

In order to maintain the uniform priority management between RT-IPC and RT-synchronization, we use a common mechanism for message queueing and priority control. The Integrated Resource Management module has been developed for this purpose[10].

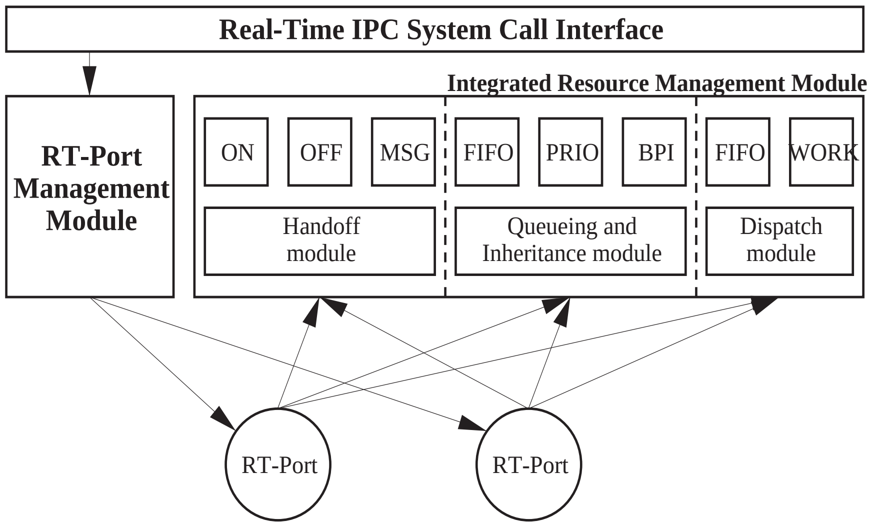
Real-Time IPC System Call Interface
Integrated Resource Management Module
RT-Port Management Module
ON
OFF
MSG
FIFO
PRIO
BPI
FIFO
WORK
Handoff module
Queueing and Inheritance module
Dispatch module
RT-Port
RT-Port

Figure 3: Real-Time IPC Structure

4.3. Integrated Resource Management Module

The basic abstraction in our model is a *resource* and a *user*. A resource represents logical resources such as a critical region or threads in a server. A user represents a schedulable entity which accesses a resource. A resource is manipulated by the two functions: *acquire* and *release*. A user must call *acquire* before using it and *release* should be called after the use. When a user tries to acquire the resource which has been acquired by other user already, the execution of the user is suspended until the resource becomes free. The suspension of the user may cause unbounded priority inversion. In our model, we assume that each user is assigned a priority using the fixed priority scheduling policy such as rate-monotonic scheduling.

The IRM Module is divided into three modules: Priority Handoff Module, Priority Queueing and Inheritance Module, and Dispatch Module. The current version of IRM Module supports the following policy modules.

Priority Handoff Module : This module supports *HANDOFF_OFF*, *HANDOFF_ON*, and *HANDOFF_MSG*. The *HANDOFF_OFF* does not change the receiver's priority. The *HANDOFF_ON* propagates the priority of sender to the receiver. The *HANDOFF_MSG* propagates the priority specified in the message header to the receiver.

Priority Queueing and Inheritance Module : This module supports *PRI_FIFO*, *PRI_PRI*, and *PRI_BPI*. The *PRI_FIFO* maintains the message queue in FIFO ordering. The *PRI_PRI* makes the message queue in priority-based order. The *PRI_BPI* makes priority ordering queue for the waiting messages, and it executes the priority inheritance protocol.

Dispatch Module : This module supports *DISP_FIFO* and *DISP_WORK*. The *DISP_FIFO* module dispatches a free resource if it is available. If not, arbitrary resource is chosen. The *DISP_WORK* selects the receiver by comparing the priority of the receivers and message.

4.4. Extended Interface

In RT-IPC *mach_port_allocate*, *mach_port_allocate_name*, and *mach_msg* cannot be used for real-time ports and real-time messages. Instead of these functions, we created *rt_mach_port_allocate*, *rt_mach_port_allocate_name*, and *rt_mach_msg* to handle real-time ports and real-time messages. Other extension is *rt_mach_port_associate* and *rt_mach_port_not_associate*. These functions are summarized as follows.

```
rt_mach_port_allocate(space, right, rt_port_attr, namep)
rt_mach_port_allocate_name(space, right, rt_port_attr, name)
rt_mach_port_associate(thread, name, priop)
rt_mach_port_not_associate(thread)
rt_mach_msg(rt_msg, option, send_size, rcv_size, rcv_name, time_out, notify)

 typedef struct {
     mach_msg_size_t     size;             /* size of buffer */
     unsigned int        nbufs;            /* Number of buffers */
     struct rt_ipc_policy {
         unsigned int    dispatch;         /* Message distribution policy */
         unsigned int    prio_inherit;     /* Priority assignment policy */
         unsigned int    prio_handoff;     /* Priority hand-off policy */
     } policy;
 } rt_mach_port_attr;

 typedef struct {
     mach_msg_bits_t     msgh_bits;
     mach_msg_size_t     msgh_size;
     mach_port_t         msgh_remote_port;
     mach_port_t         msgh_local_port;
     mach_port_seqno_t   msgh_seqno;
     mach_msg_id_t       msgh_id;
     rt_priority_data_t  msgh_priority;    /* message priority */
 } rt_mach_msg_header_t;
```

When a real-time port is allocated, *rt_mach_port_attr* specifies port policies (i.e., handoff, queueing, and dispatch policy), number of kernel message buffers and its size. *rt_mach_msg_header_t* has the same field of the original Mach message header. When HANDOFF_MSG is chosen as the Handoff policy, *msgh_priority* is added to the real-time message header to specifies a message priority.

rt_mach_port_allocate and *rt_mach_port_allocate_name* allocate a real-time port or a port set. The difference between two is whether the port name is given by the system or user.

Before receiving messages, each receiver thread must be associated with a real-time port or port set by calling *rt_mach_port_associate*, and then the thread is recognized as a server thread. In the current implementation, threads can be associated with one port or port set. When DISP_WORK is selected as Dispatch policy, *priop* in the arguments specifies the priority which is used the receiver priority. If a thread tries to associate with a normal port or port set, then *rt_mach_port_associate* returns with KERN_INVALID_NAME.

rt_mach_port_not_associate breaks an association between a receiver thread and the real-time port or port set. After it calls this primitive, the thread cannot receive from the real-time port or port set unless calling *rt_mach_port_associate* again.

rt_mach_msg sends and/or receives a real-time message. The arguments of this function are the same as *mach_msg* which is the original Mach IPC send and receive primitive, except the option argument and the header of the message. If the message is sent in synchronous transmission mode, the MACH_SEND_SYNC flag must be specified in the option argument. If a user tries to send a message to a normal port using *rt_mach_msg*, then the user get an error of MACH_SEND_INVALID_DEST. The Priority inheritance and priority hand-off are in affect until the receiver issues a next message receive call.

Other port manipulation primitives, such as *mach_port_insert_right*, *mach_port_deallocate*, and *mach_port_destroy* can also be used for the real-time port.

5. Performance Evaluation

We measured the transmission cost with different policies to evaluate the low-level RT-IPC primitive cost. We also executed a Distributed Hartstone benchmark program for high-level performance evaluation to compare the effectiveness of various policies.

In the following evaluation, we compared the cases where *Mach IPC*, RT-IPC with *FIFO/OFF*, *PRIO/ON*, and *BPI/ON* policies are used. *Mach IPC* indicates the original Mach IPC, *FIFO/OFF* means measured with FIFO queueing without priority handoff. *PRIO/ON* represents priority-based message queueing with priority handoff. *BPI/ON* means basic priority inheritance with priority-based queueing and priority handoff enabled.

We used a Gateway 2000 4DX2/66E which has a 66MHz Intel 80486DX2 processor and 16 megabytes of memory. The system was running RT-Mach version MK78 with CMU UNIX server version UX39 in a single user mode. Additionally, the network was disconnected and the benchmarks were created with the highest thread priority. We used an Alpha Logic's STAT! timer board to take measurements accurate to 250 nanoseconds.

5.1. Message Transmission Cost

The basic transmission cost of RT-IPC in Figure 4 and 5. The measurement was repeated 1000 times and the averages are taken. Figure 4 shows the round trip transmission cost of RT-IPC and the original Mach IPC where a receiver is ready and waiting for a message. The costs of sending a message while the receiver is servicing another request are shown in Figure 5.

The cost of RT-IPC is 10 to 20% higher than that of Mach-IPC, because of the additional functionality. The differences among the various policies of RT-IPC are due to the different characteristics of the message queueing and priority control policies. The gap between RT-IPC and Mach-IPC are different where the message size is less than 512 bytes and over 512 bytes. This comes from the different IPC path in the microkernel and cache effects. When a message is large, Mach-IPC calls some routines to allocate and deallocate a kernel message buffer, but the RT-IPC does not need this, since the message buffers are allocated previously.

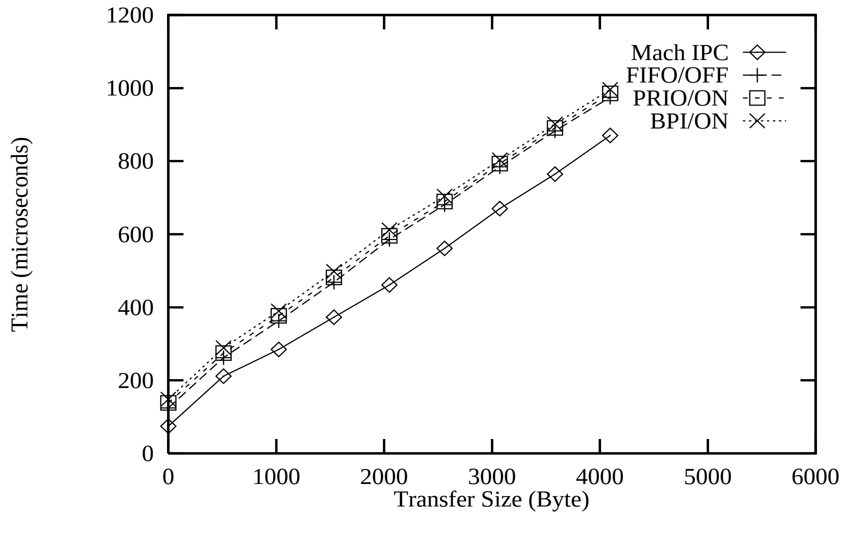

Figure 4: Round Trip Cost

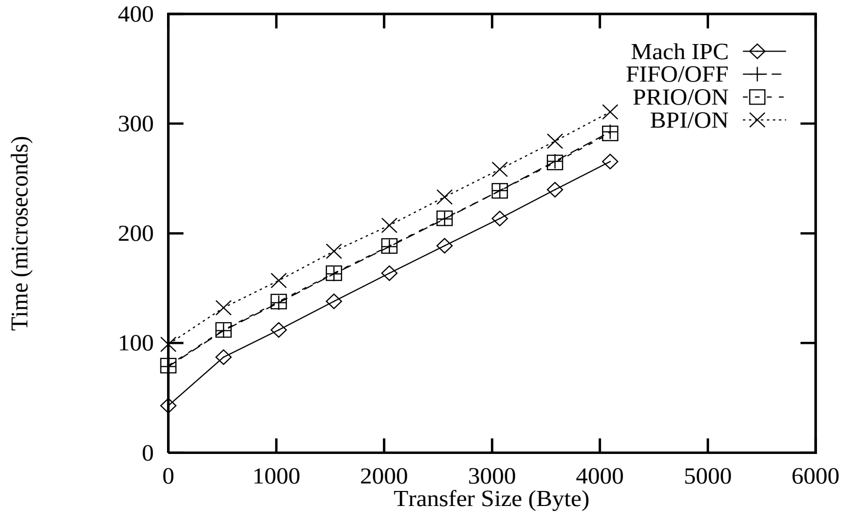

Figure 5: Message Send Cost when Receiver is Busy

5.2. Distributed Hartstone Benchmark

Distributed Hartstone (DHS)[7] is a series of extended Hartstone real-time benchmark programs and is designed for evaluating distributed real-time systems. Since the benchmark is designed for the distributed environment, the original DHS requires a set of servers and clients running on different machines connected by a network. In this evaluation, we executed the servers and clients on a single machine in order to evaluate the local IPC performance.

DSHcl is a Distributed, Synchronized, and Harmonic task set which originally tests the communication latency of the system. In this study, we expect this benchmark to evaluate the effect of priority assignment as well as communication latency. There are five client threads τ_1, ..., τ_5 which are periodic threads. Each of the client threads sends a request to the server before consuming its own workload. The benchmark uses the kilo-Whetstone as a unit of computation derived from the popular Whetstone benchmark. The server τ_{server} is an aperiodic thread. We vary the computation time of the server to gradually increase until any client thread misses a deadline. Figure 6 shows the structure of the benchmark. The timing requirements for each threads are shown in Table 1.

DHSpq is a Distributed, Synchronized, and Harmonic task set which is designed to test for priority queueing for communication packets. We expect this test to show the advantage of priority queueing over the conventional FIFO queueing. Like DSHcl benchmark, DSHpq has five clients and one server thread. The DSHqp benchmark is quite similar to the DSHcl except for the difference in granularity. Table 1 illustrates the timing requirement of the threads.

Table 2 summarizes the result of DHS benchmark. The numbers in the table represent the breakdown CPU utilization and the larger utilization number indicates the better system services.

The both results of the benchmarks show that RT-IPC provide higher CPU utilization comparing with Mach IPC. In case of *BPI/ON*, CPU utilization is slightly lower than that of *PRIO/ON*. This is caused by the overhead of priority inheritance.

6. Related Work

There has been a discussion and attempt of real-time IPC extension for multimedia systems on microkernel, however it was not realized before[17].

Many commercial and research operating systems provide some real-time features. For example,

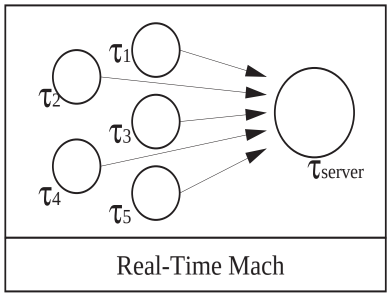

Figure 6: Distributed Hartstone Benchmark

Task	Workload	Period	
		DHScl	DHSpq
τ_1	1 KWS	80 ms	160 ms
τ_2	1 KWS	160 ms	320 ms
τ_3	2 KWS	320 ms	640 ms
τ_4	2 KWS	640 ms	1280 ms
τ_5	8 KWS	1280 ms	2560 ms
τ_{server}	variable	N/A	N/A

Table 1: DHS Task Set

	DSHcl	DHSpq
Mach IPC	60.6	55.3
FIFO/OFF	60.6	55.3
PRIO/ON	94.0	90.5
BPI/ON	93.4	88.3

Table 2: DHS result

Chorus[13] microkernel provides priority-based preemptive scheduling and fast interrupt latency. However, it does not provide the mechanisms to avoid unbounded priority inversion.

Lynx operating system[4] provides the priority inheritance protocol for real-time synchronization. However, the Lynx IPC does not provide priority inheritance for IPC, and cannot mix these two features without having priority inversions. POSIX[11] also proposed priority inheritance, but it does not discuss about the integration of IPC and synchronization.

There is a similar mechanism provided by QNX[5]. It provides priority-based message queueing and priority management mechanisms for IPC. However, it only supports synchronous communication mode, and is not integrated with the real-time synchronization facility.

A real-time extension of Mach for multimedia applications has been proposed[8]. It provides asynchronous event notification, preemptive deadline-driven scheduling, and temporal paging system. However, real-time extension of IPC nor Synchronization were not supported.

For real-time synchronization, the Stack Resource Policy (SRP)[1] has been proposed. It is also a starvation free locking protocol. It has been implemented only for real-time synchronization.

7. Conclusion

In this paper, we demonstrated the RT-IPC model, its implementation and its performance. Although the message transmission cost of RT-IPC is about 10 to 20% higher than the original Mach IPC, the DHS benchmark results indicate that RT-IPC significantly reduces priority inversions and improves the breakdown CPU utilization. We are still investigating the overhead of transmission cost and will reduce the gap between the Mach IPC and RT-IPC cost. Each policy has different cost and characteristic, a system designer needs to select a proper policy for the application program to minimize priority inversion problem and maximize CPU utilization.

RT-IPC has been used for Real-Time Server (RTS) and the Network Protocol Server (NPS)[9], and both servers successfully improve the breakdown CPU utilization of real-time applications.

Acknowledgments

We would like to thank the members of the ART Project and the Mach group for their valuable comments and inputs to the development of Real-Time Mach.

References

[1] T.P. Baker. A Stack-Based Resource Allocation Policy for Realtime Processes. In *Proceedings of IEEE 11th Real-Time System Symposium*, December 1990.

[2] R.P. Draves. A Revised IPC Interface. In *Proceedings of USENIX 1st Mach Workshop*, October 1990.

[3] R.P. Draves, B.N. Bershad, R.F. Rashid, and R.W. Dean. Using Continuations to Implement Thread Management and Communication in Operating System. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, October 1991.

[4] B.O. Gallmeister and C. Lanier. Early Experience with POSIX 1003.4 and POSIX 1003.4A. In *Proceedings of IEEE 12th Real-Time System Symposium*, December 1991.

[5] D. Hildebrand. An Architectural Overview of QNX. In *Proceedings of Workshop on Micro-Kernel and Other Kernel Architectures*, April 1992.

[6] K. Loepere. *Mach 3 Kernel Principles.* Open Software Foundation and Carnegie Mellon University, 1992.

[7] C. W. Mercer, Y. Ishikawa, and H. Tokuda. Distributed Hartstone: A Distributed Real-time Benchmark Suite. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, May 1990. Also available as Technical Report CMU-CS-90-110.

[8] J. Nakajima, M. Yazaki, and H. Matsumoto. Multimedia/Realtime Extension for Mach 3.0. In *Proceedings of Workshop on Micro-Kernel and Other Kernel Architectures*, April 1992.

[9] T. Nakajima, T. Kitayama, and H. Tokuda. Experiments with Real-Time Servers in Real-Time Mach. In *Proceedings of 3rd USENIX Mach Symposium*, April 1993.

[10] T. Nakajima, T.Kitayama, H. Arakawa, and H. Tokuda. Integrated Management of Priority Inversion in Real-Time Mach, 1993. The ART group Tech. Memo.

[11] *IEEE Standard P1003.4 (Real-time extensions to POSIX).* IEEE, 345 East 47th St., New York, NY 10017, 1991.

[12] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, 1991.

[13] M. Rozier, V Abrossimov, F. Armand, I Boule, M Gien, M Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems Symposium, 1990. Chorus syste'mes, Technical Report CS-TR-90-25.

[14] S. Savage and H. Tokuda. RT-Mach Timers: Exporting Time to the User. In *Proceedings of USENIX 3rd Mach Symposium*, April 1993.

[15] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. Technical Report CMU-CS-87-181, Carnegie Mellon University, November 1987.

[16] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[17] J.A. Test. Mach 3.0 Multimedia Real-Time Requirements, 1992. Proceedings of OSF Research Institute Symposium.

[18] H. Tokuda and T. Nakajima. Evaluation of Real-Time Synchronization in Real-Time Mach. In *Proceedings of 2nd USENIX Mach Workshop*, November 1991.

[19] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proceedings of USENIX 1st Mach Workshop*, October 1990.

Takuro Kitayama is a Visiting Scientist in the School of Computer Science at Carnegie Mellon University. He is originally from Oki Electric Industry Co., Ltd. where he is employed since 1986. His research interests is real-time operating systems and multimedia operating systems. He received BS degree in engineering from Kougakuin University in 1986. He is a member of the ACM.

Tatsuo Nakajima is an Associate Professor of Information Center at Japan Advanced Institute of Science and Technology, where he is employed since 1993. Prior to that he worked on Real-Time Mach at Carnegie Mellon University. He received his PhD from Keio University in 1990 in distributed reliable computing. His research interest is operating systems for multimedia, high performance operating systems, reliable communications, and object-oriented languages. He is a member of the ACM and the Japan Society for Software Science and Technology.

Hideyuki Tokuda is a Senior Research Computer Scientist in the School of Computer Science at Carnegie Mellon and also an Associate Professor in the Faculty of Environmental Information at Keio University. His research interests include distributed real-time systems, multimedia systems, communication protocols, and massively parallel/distributed systems.

Tokuda received BS and MS degrees in engineering from Keio University and a PhD degree in computer science from University of Waterloo. He is a member of the IEEE, the ACM, the IPSJ, and the Japan Society for Software Science and Technology.

Availability

Mach 3.0 microkernel is free and available for anonymous FTP from cs.cmu.edu.

For RT-Mach, we distribute binaries for anonymous FTP from k.gp.cs.cmu.edu. The distribution includes the RT-Mach microkernel, libraries, and header files. The detailed description of the files are stored in /usr/arts/public/rtmach/README. You need to have an original Mach 3.0 environment before getting RT-Mach.