Modeling and Validation of the Real-Time Mach Scheduler

Hiroshi Arakawa, Daniel I. Katcher, Jay K. Strosnider, Hideyuki Tokuda
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
katcher@ece.cmu.edu

Abstract

Real-time scheduling theory is designed to provide *a priori* verification that all real-time tasks meet their timing requirements. However, this body of theory generally assumes that resources are instantaneously preemptable and ignores the costs of systems services. In previous work [1, 2] we provided a theoretical foundation for including the costs of the operating system scheduler in the real-time scheduling framework. In this paper, we apply that theory to the Real-Time (RT) Mach scheduler. We describe a methodology for measuring the components of the RT Mach scheduler in user space. We analyze the predicted performance of different real-time task sets on the target system using the scheduling model and the measured characteristics. We then verify the model experimentally by measuring the performance of the real-time task sets, consisting of RT Mach threads, on the target system. The experimental measurements verify the analytical model to within a small percentage of error. Thus, using the model we have successfully predicted the performance of real-time task sets using system services, and developed consistent methodologies to accomplish that prediction.

1 Introduction

In real-time computing, correctness depends not only on the results of computation but also the time at which outputs are produced and communicated. A primary goal of real-time systems design is to ensure timing correctness of real-time tasks. To achieve this goal, significant work has gone into developing scheduling algorithms and analysis that determine *a priori* if a given set of tasks will meet their timing requirements on a given set of resources. However, there is a significant gap between the scheduling theory that verifies that all tasks will meet their timing requirements and the implementation of real-time scheduling algorithms on systems consisting of real processors with real operating systems and real networks. This theory must be updated to reflect the costs of implementation.

The first step towards closing that gap is to account for the costs of operating system services. We wish to develop a methodology for characterizing (measuring) real-time kernels and a theory to allow the user to include those measurements in the scheduling analysis framework.

Unfortunately, the current state of the practice in commercial real-time operating systems is not useful to allow real-time systems designers to include these costs in their systems analysis. Currently, vendors quote performance numbers for isolated measurements that are meaningless from a scheduling theory standpoint. Table 1 shows a list of commonly quoted numbers pulled from recent discussions on a real-time bulletin board [3]. Though interesting for a first-level comparison of kernel performance in isolated areas, these measurements hold little value because they do not tell the systems designer what he or she needs to know: what are the true costs of using a given operating system, how do I include these costs in a scheduling analysis framework, and will my tasks meet their timing requirements if I use this operating system and its services.

	pSOS+	VRTX32	LynxOS	VxWorks	PDOS
Create/Delete Task	591	371	–	1423	1113
Ping Suspend	114	128	–	117	79
Suspend/Resume	71	83	–	69	27
Get/Release Semaphore	55	63	55	74	2
Interrupt Response	6	6	13	6	3
Int/Task Response	163	169	175	125	41

All times in $\mu secs$

Table 1: Commonly Quoted Performance of Commercial Kernels

In previous work [1, 2] we developed scheduling models for different theoretical implementations of operating system schedulers. The scheduler is the core of the operating system, and only after including its costs can we hope to include other system services. This theory [1] included the costs of interrupt handling, scheduling, and preemption to allow the user to analyze the schedulability of a set of periodic real-time tasks with hard deadlines. This paper applies that theory to develop a scheduling model for the Real-Time Mach fixed priority scheduler. We characterize the RT Mach scheduler in the *worst case* using the *pixie* [4] profiling tool. Worst case measurements must be used to ensure that timing correctness is maintained at all times. We develop analytical models for real-time task sets' performance using the RT Mach scheduling model and the measured RT Mach components. Finally, we verify the RT Mach scheduling model experimentally against the analytical models on a DECstation 5000 running the RT Mach kernel using real-time threads with tightly controlled timing characteristics. These results show that we are able to predict the timing performance of real-time task sets on the target platform to within a small percentage error.

The paper is organized as follows. Section 2 provides background in real-time scheduling theory and analysis. In Section 3 we develop a model for the Real-Time Mach scheduler. Section 4 characterizes the MK75 version of RT Mach running on a DECstation 5000. This section includes a methodology for performing this characterization using *pixie* and the results of the measurements. The model is validated experimentally in Section 5 by measuring the breakdown utilization of a set of real-time threads on the target system. Section 6 discusses our conclusions and future directions for this work.

2 Background

We first present a formal notation for describing a real-time task set. We consider here only "hard" periodic real-time tasks, where each task has a deadline by which its execution must be complete, otherwise the system is said to have failed. Aperiodic tasks have been included within the periodic scheduling framework through the use of aperiodic servers in [5, 6]. Other modifications to the original periodic scheduling framework include adding synchronization [7] and scheduling for buses [8]. A good overview of real-time systems is found in [9].

A real-time task set is composed of n tasks. Each task τ_i, $1 \leq i \leq n$, is described by a period, T_i, a deadline, D_i, and a worst case estimated execution time, C_i. We assume that all deadlines are before or at task periods. Each instantiation of a task, which occurs once per period, is called a job of that task. Tasks are scheduled by fixed priority, preemptive algorithms, which assign the resource to the task with the highest priority. A task set that meets all its deadlines is said to be schedulable.

Liu and Layland developed *sufficient* conditions for determining if a task set is schedulable under any fixed priority algorithm based on its utilization. Utilization is defined as $U = \sum_{i=1}^{n} C_i/T_i$. They found that a task set is schedulable if $U \leq n(2^{1/n} - 1)$. This bound rapidly converges to $ln\ 2 = 69\%$ as n gets large. However, the Liu and Layland bound is just sufficient; a task set can have a higher utilization and still be schedulable. The bound developed by Lehoczky, Sha, and Ding [10] provides a *necessary and sufficient* condition for schedulability. This bound checks the worst case response time of every task. The worst case is found for a *critical instant*, where every other higher priority task arrives simultaneously with the task in question. If each task meets its deadline in its first period following a critical instant, then all tasks will always meet their deadlines under any phasing. This is expressed mathematically as:

$$\forall i, \quad 1 \leq i \leq n, \quad \min_{0<t\leq D_i} \left(\sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \right) \leq 1$$

Both the Liu and Layland and the Lehoczky, Sha, and Ding scheduling tests are founded on the critical assumptions that the resource is instantaneously preemptable, and that preemption incurs no cost. In reality, task sets on a CPU are implemented via an operating system, and the operating system incurs cost to run scheduling code, handle interrupts, and perform context switches. Further, many operating systems are non-preemptable. Harbour, Klein, and Lehoczky [11] developed a test for fixed priority scheduling under non-preemptable conditions. In previous work [1, 2] we built upon [11] and [10] to develop generic scheduling models for different types of operating system scheduler implementations. We will briefly discuss this work to explain the costs of implementing a real-time task set via an operating system scheduler.

We defined two basic means of implementing a scheduler: event driven and timer driven. An event driven scheduler uses interrupts from some hardware device that coincide with the start times of the periodic tasks. Every time a task is to be initiated, an interrupt arrives signalling this event. This implementation is most often found in signal processing environments, or other data driven systems. Timer driven systems rely on periodic timer interrupts to allow processing to be interrupted so the scheduler can run. On every timer interrupt, the scheduler updates the internal system time and initiates tasks whose start times have passed. In addition, we differentiated systems by the preemptability of the kernel. An ideal kernel would always provide service at the priority level of the requesting task, so that if a higher priority request arrived, it would immediately be serviced. In this case, high priority tasks never have to wait for lower priority tasks to execute. However, in reality, many operating systems are non-preemptable, and must complete a given service once it has begun. This is done in practice to limit the complexity of the internal kernel code and data structures.

The RT Mach operating system scheduler is a timer driven, non-preemptable implementation. We will next describe the details of a generic timer driven system, from [1]. We assumed that there is a periodic timer that interrupts processing every T_{tic} seconds, allowing the scheduler to be invoked. Costs are categorized as either *overhead* or *blocking*. Overhead is cost that is directly incurred by the task requiring service. Blocking is cost required to service or execute a lower priority task, and is commonly referred to as *priority inversion* [7].

We defined the following overhead and blocking terms:

- C_{timer} is the time required to handle a clock interrupt. The C_{timer} term is overhead that occurs on every timer interrupt, and includes the time to handle the interrupt, update the internal system time, and call the scheduler.
- $C_{preempt}$ is the time for scheduling and preemption for one task, assuming that it is higher priority than the active task. This is treated as overhead for the preempting task and includes scheduling and a context switch.
- $C_{nonpreempt}$ is the time for scheduling without preemption for one task, assuming that it is lower priority than the active task. This term shows up as blocking for higher priority tasks.
- C_{exit} is the time it takes for a task to exit when it completes, which is considered overhead for that task.
- C_{system} is the longest non-preemptable segment of the operating system, and is considered blocking for every task in the set. We can assume for our analysis that this segment is initiated by a non-real-time, background task.

We restate a theorem from [12]:

Theorem 1 *For a non-preemptable timer-driven scheduler, a set of n tasks* $\tau_1, ..., \tau_n$ *is schedulable for all task phasings if the following conditions hold:*

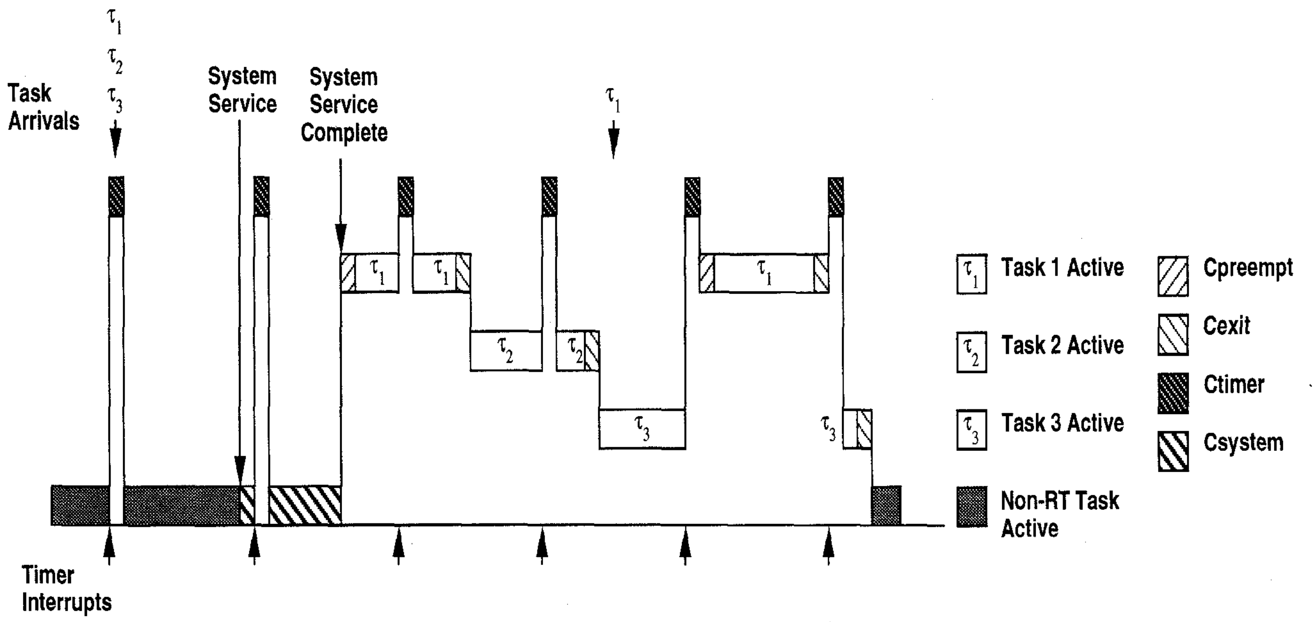

Figure 1: Worst Case Scheduling in RT Mach

$$\forall i, \quad 1 \leq i \leq n, \quad \min_{0<t\leq D_i}$$
$$\{\left(\sum_{j=1}^{i} \frac{C_j+C_{preempt}+C_{exit}}{t}\lceil\frac{t}{T_j}\rceil\right) +$$
$$\lceil\frac{t}{T_{tic}}\rceil\frac{C_{timer}}{t} + \sum_{j=i+1}^{n}\lceil\frac{t}{T_j}\rceil\frac{C_{nonpreempt}}{t} +$$
$$\frac{T_{tic}}{t} + \frac{C_{system}}{t}\} \leq 1$$

This theorem changes the Lehoczky, Sha, and Ding criterion to a sufficient condition for scheduling by adding in the costs of implementation to extend the worst case response time of each task starting from a critical instant. It is sufficient because the worst case overhead and blocking costs may never occur in practice. The theorem adds in the overhead of preemption and exiting to each task's execution time and adds the overhead of the timer interrupt service. It also extends the response time of each task by adding in the blocking due to every lower priority task that must be scheduled, the blocking due to preemption limited by the timer granularity, and the blocking due to the non-preemptable system service, C_{system}. We previously defined T_{tic} as the interval between timer interrupts. This shows up in the scheduling equation as a blocking term, because in the worst case a high priority task's critical instant will begin just after a timer interrupt arrives, causing the response time to be delayed by one timer interrupt interval. Figure 1 shows the worst case for three real-time tasks.

In the next section we will apply this theorem to the Real-Time Mach scheduler and show how the components of RT Mach map into the terms defined above.

3 Real-Time Mach Scheduler Model

In this section we will describe the Real-Time Mach scheduler implementation and present a scheduling model for the RT Mach scheduler. RT Mach [13] was developed at Carnegie Mellon University as part of the Advanced Real-Time (ART) project. It is based on the Mach kernel [14, 15] but augments the structure of the scheduling and interprocess communications (IPC) functions to support real-time scheduling, synchronization, and communications.

RT Mach is a microkernel operating system; it provides the core functions of an operating system. More complex OS services, such as those found in monolithic kernel structures, can be implemented via an OS server on top of a microkernel. An overview of this structure is pictured in Figure 2. The microkernel structure is beneficial to the cause of real-time predictability, because it allows us to easily isolate and measure the different components of the operating system. Then, when these underlying components are fully characterized, we will be able to characterize higher level features such as file management, database management, and window systems.

RT Mach has *thread* and *task* abstractions, where

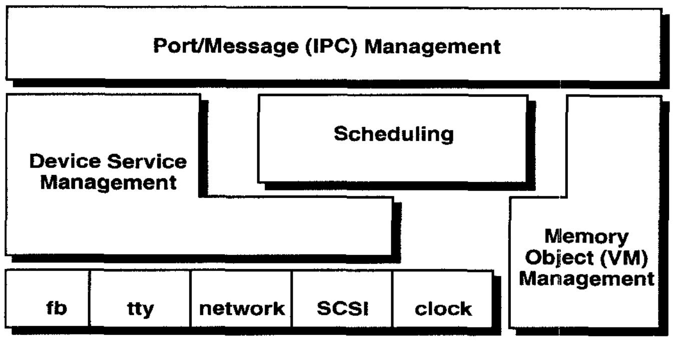
Port/Message (IPC) Management
Device Service Management
Scheduling
Memory Object (VM) Management
fb
tty
network
SCSI
clock

Figure 2: Overview of the Real-Time Mach Microkernel

a thread is a unit of execution and a task is a unit of resource allocation. Only threads execute; they are resources associated with a given task. In the following analysis we will consider a real-time task set to consist of multiple RT Mach threads within one RT Mach task. Therefore, context switching always occurs within a single address space. Context switching between RT Mach tasks requires a far more expensive cross-address space switch and is not considered here.

The RT Mach scheduler supports periodic, aperiodic, and non-real-time task scheduling and has a queue structure that supports aperiodic and non-real-time (background) processing. The queues we are concerned with are the *start* queue, which holds threads that are waiting for their start times to expire, and the hard real-time (HRT) *run* queue. There are actually 32 doubly linked run queues, but the rest are dedicated to aperiodic and soft real-time tasks.

As we mentioned, RT Mach is entirely non-preemptable. This means that in our worst case analysis, we have to consider the case where there is a request for some RT Mach service immediately prior to the critical instant. This request can come on behalf of a non-real-time task. In the worst case, this time is maximized when the service is the longest possible RT Mach system service. For the purposes of this analysis, we map the term C_{system}, which is the longest possible non-preemptable system service, to C_{exit}. Ultimately, we hope to determine the longest possible system segment, and include that in our model.

The four components to be measured, C_{timer}, $C_{preempt}$, $C_{nonpreempt}$, and C_{exit}, must be measured in the worst case. If we set up the worst case properly, then we can guarantee that if a task set is schedulable under these worst case conditions, then it will always be schedulable. C_{timer} is constant; it is the time to only handle timer interrupts, without any scheduling. The other three components are all functions of the number of threads in the system. Specifically, $C_{preempt}$ moves a single thread from head of the start queue to the run queue and preempts the active task, which must then be stored in the run queue. For our worst case analysis, we have to assume that every other thread in the system is in the run queue. The thread being scheduled has the highest priority and gets inserted at the head of the run queue. However, in the worst case we must assume that the task being preempted is inserted at the tail of the run queue, requiring a search of the entire queue. In steady state, this case will seldom occur, but we are trying to ensure timing correctness in the absolute worst case. Similarly, for $C_{nonpreempt}$, which moves a lower priority thread from the start queue to the run queue without a context switch, we assume that every other thread is in the run queue and that the scheduled thread gets inserted at the tail of the run queue. Finally, C_{exit} moves a completing thread into the start queue and then selects the head of the run queue to run. In the worst case, the exiting task will get inserted at the tail of the start queue and every other thread in the system will be in the start queue. Note again that all components, except for C_{timer}, are a function of the number of threads in the system.

These theoretic components, $C_{preempt}$, $C_{nonpreempt}$, C_{exit}, and C_{timer}, correspond functionally to calls to three RT Mach kernel primitives: *clock_interrupt*, *thread_block*, and *thread_exit*. The interrupt handler *clock_interrupt* is invoked on every timer interrupt. It handles the actual interrupt, updates the system time, and performs the scheduling functions of updating the start and run queues. *thread_block* handles context switching. *thread_exit* is similar to *thread_block*, except for the destination of the exiting thread, which is stored back on the start queue.

Each of these routines will be measured, along with the associated low level interrupt handling and resumption code, in the worst case scenarios described above.

For C_{timer}, the time to run the interrupt handler only, we measure *clock_interrupt* with no threads in the start queue, so that the primitive is invoked and then exits without any queue manipulations. To measure $C_{preempt}$ for n threads, first the highest priority thread must be at the head of the start queue, $n - 2$ lower priority threads must be in the run queue, and the lowest priority thread in the system must be the active thread. Then *clock_interrupt* and *thread_block* are invoked and measured. Likewise, $C_{nonpreempt}$ corresponds to the case where $n - 2$ threads, all of higher priority, are in the run queue when *clock_interrupt* is invoked. Finally, $n - 1$ threads with earlier start times than the exiting thread must be in the start queue, when *thread_exit* is measured.

The following summarizes how the terms defined in the RT Mach scheduler model correspond to the calls to the RT Mach routines.

- C_{timer} : *clock_interrupt* with no threads on the start queue, so that no scheduling is performed.
- $C_{preempt}$: *clock_interrupt*, but first sets up the worst case scenario described above. For this measurement, *clock_interrupt* also calls *thread_block* to perform the context switch.
- $C_{nonpreempt}$: *clock_interrupt*, but first sets up the worst case scenario described above, but no context switch occurs.
- C_{exit} : corresponds directly to *thread_exit* with the worst case scenario described above.

Again, these worst case scenarios may seldom occur in practice. But we are trying to supply a solid sufficient condition, such that a set of real-time threads said to be schedulable will indeed always be schedulable when implemented.

4 Characterization of the Scheduler

This section will describe the methodology and results of characterizing the RT Mach scheduling model. Characterization is the measurement of the components of the RT Mach scheduling equation in the worst case scenarios described above. The vendor benchmarks from Table 1 supply measurements that can be accomplished directly from user mode, but we are really interested in measurements that require initial setup of the kernel queues to reflect the worst case placement of the threads in the system. This is very difficult, because setting up the worst case scenarios requires direct access to the kernel queues, which are privileged, and is not possible with calls from user mode.

Our solution to this problem is to compile kernel code as a user process and then measure the performance of the components using *pixie* [4]. This approach gives us the dual advantages of being able to initialize the worst case scenarios correctly and then measure kernel code very accurately without any additional overhead. We will next describe this process in more detail.

The *pixie* tool adds code to user applications to count such things as the number of times procedures or basic blocks are run, and the number of cycles that each takes. The user then runs this augmented code, and uses the *prof* tool to generate profiling data based on the output of the *pixie* code. For our measurements these two tools are run, then we perform post processing to extract the final cycle counts for the components we wish to measure. This process is shown in Figure 3.

However, to compile these tools with kernel code in user space, we must use special stub routines to simulate privileged kernel code and to initialize kernel data structures, such as the scheduling queues. Privileged code is required to perform machine-dependent work, such as low level interrupt handlers, context switch code, or a change in the processor interrupt mask. These routines must be simulated with stub routines that emulate the actual processing. Further, the number of cycles of the privileged routines is manually counted; these numbers are inserted in the post processing phase of the measurement. The initialization stub routines are required to set up the state of the kernel and the scenario. These routines initialize the scheduling and IPC queues, create threads, initialize other miscellaneous data structures, and call the actual routines that get measured.

The initialization routines first set up the worst case scenarios described in the previous section, then call the kernel routines. These routines are measured in the post processing phase, which adds in the cycles counted man-

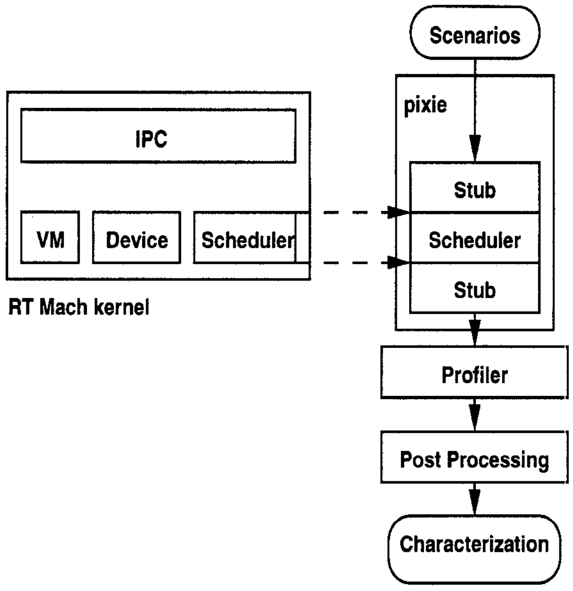

Figure 3: Characterization Process

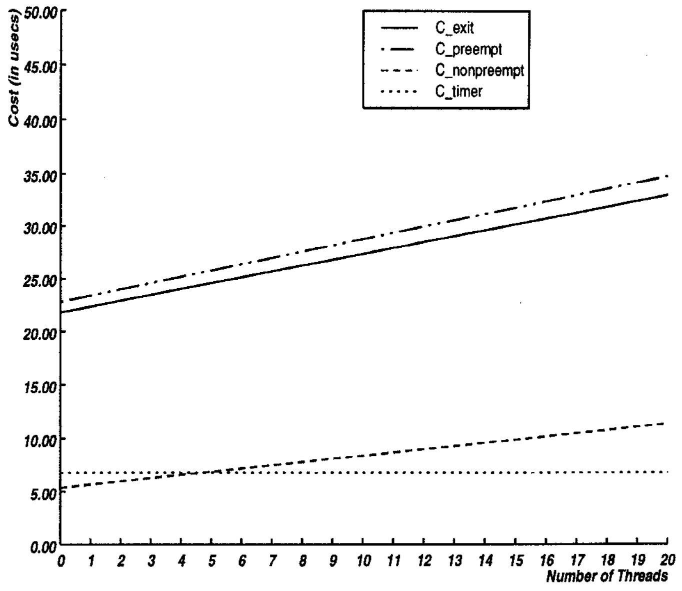

Figure 4: Costs of Kernel Components

ually for kernel code which must be stubbed out. *pixie* supplies the cycle counts for all other code. The scenarios, and measurements, were repeated with the number of threads varying from 0 to 20. The measurements were performed on a DECstation 5000/200 25 MHz MIPS R3000. The results were linear, and thus not repeated beyond 20 threads in the system. Results for all four components are shown in Figure 4. For example, for 10 threads, $C_{timer} = 7.92\ \mu secs$, $C_{preempt} = 38\ \mu secs$, $C_{exit} = 36.2\ \mu secs$, and $C_{nonpreempt} = 11.02\ \mu secs$. These results correspond to the following, where n is the number of threads in the system:

$$C_{timer} = 7.92\ \mu secs$$

$$C_{preempt} = 0.79n + 30.1\ \mu secs$$

$$C_{exit} = 0.74n + 28.8\ \mu secs$$

$$C_{nonpreempt} = 0.39n + 7.12\ \mu secs$$

We now have a scheduling model for the RT Mach scheduler, with the components of that model measured in the worst case on the target platform. The next step is to use this model to make predictions for real-time task sets' timing performance. To do this we create a set of synthetic threads on the target platform and measure their timing performance. The next section describes this process.

5 Validation of the Model

In this section we will show the methodology and results of validating the RT Mach scheduler model developed in Section 3 using the measurements described in Section 4. We first describe the experimental validation method, which involves creating synthetic task sets with very tightly controlled timing requirements and gradually scaling their run times until a missed deadline is detected. We also scaled each task set to their *breakdown utilization* using the RT Mach scheduling model. Finally, the breakdown utilization measured experimentally is compared to the analytical results found from the RT Mach scheduling model and the worst case characterization.

Breakdown utilization [10] is found by uniformly scaling all the execution times, C_i, while holding the periods fixed, until the task set is just schedulable. It is defined as $U^* = \sum_{i=1}^{n} \frac{\alpha C_i}{T_i}$, where α is a scaling factor that is applied uniformly to all the run times in a task set, while the periods remain fixed. Specifically, U^* is the utilization at which all deadlines are met, but any further increase in α will cause one or more tasks to miss their deadlines. Breakdown utilization can be thought of as a measure for the amount of tolerance to low execution time estimates for a task set, or simply as a measure of the maximum utilization possible for a given task set.

The experimental measurement of breakdown utilization is accomplished by creating a task set on the target system consisting of real-time threads which have very tightly controlled execution times. The execution times of these threads are then gradually scaled until at least one of them misses a deadline. The threads are run for a sufficient amount of time at each scaling level to verify that the task set is schedulable. After executing for a sufficient time, we scale the run times of the threads up and re-run the task set. With this scaling code we can exactly scale the task sets to their breakdown utilization on the target system. The utilization at the point where a thread misses a deadline is recorded and compared to the analytical results. The RT Mach kernel has a facility for detecting missed deadlines. When a thread is initiated, a timer is set for the thread's deadline. If the timer expires before the thread exits, then the deadline has been missed and the kernel sends a message to an error handling routine that is specified by the user.

The synthetic threads that represent the task sets are all identical and are created with known execution times determined with *pixie*. The following periodic thread body consumes CPU time as defined in a global **run_times** array. The address reference in the body of the loop ensures that the loop code is executed, and not optimized away by the compiler. The loop is four cycles long; we subtract the cost of the call to *which_thread*, which returns the current thread id. The **run_times** array is scaled to increase the execution time of the threads. This code will ensure that the execution time is exact.

```
periodic_thread_i()
   {
   int i, *k, thread, count;
   /* get current thread id */
   thread = which_thread(mach_thread_self());
   /* get loop count from run_time and loop */
   count = ((run_times[thread]*25)/4)-50;
   for (i=0; i<count; i++)
      k = &i;
   }
```

We will next describe several different task sets, the analytical results derived for each, and compare those results to the experimental measurements performed on the target system. We executed each task set at each scale level for the duration of the hyperperiod of the task set. The hyperperiod is the least common multiple of all the tasks' periods; from any initial phasing it is the point in time where the schedule is guaranteed to repeat. Each measurement was repeated at least five times and averaged. Generally, the results were very consistent, with little variance, but differences are noted below. Measurements were conducted on the target 25 MHz MIPS R3000 processor. Unfortunately, we were unable to perform these measurements on a target isolated from the local network. The target was connected to an Ethernet network, and measurements were repeated for both weekday (heavy) and weekend (light) loading. Because of heavier network interference, the maximum schedulable utilization was decreased roughly two to four percent during weekday hours. All figures shown use lightly loaded measurements.

For each task set, scaling was done over a range of timer interrupt intervals, to examine the effect of the timer interrupt rate on schedulability. We analytically determined the breakdown utilization over a range of timer interrupt values using the RT Mach scheduling model. We then measured the breakdown utilization on the target platform for several different timer interrupt intervals. RT Mach on the target platform allows the timer interrupt interval to be configured, but only to either 1, 2, 4, 8, or 16 msecs. Figure 5(a) shows these results.

The avionics task set [16, 17] is representative of a real-time system used for mission control on an airplane. The avionics task set consists of 15 periodic tasks, with periods ranging from 25 msecs to one second. The utilization of the task set, without overhead or blocking costs, is 83.01%. We first analytically scaled the avionics task set to its breakdown utilization using the measured characteristics from Section 4. The first important point of this figure is the significant difference between the ideal breakdown utilization, calculated using the original Lehoczky, Sha, and Ding [10] scheduling equation, and the RT Mach scheduling model breakdown utilization, calculated using the methods of this paper with overhead and blocking. For example, att the 8 msec timer interrupt interval, there is an 11% difference between the ideal and analytical points of 92.23% and 83.84%, respectively.

Second, we note that the measured results on the target system were within several percentage points of the analytical results. The exception to this is at the 16 msec timer interrupt interval. There the measured breakdown utilization was 82.16% and the breakdown utilization predicted by the model was 72.61%. There are several reasons for this difference, most important of which is a weakness in the current method of deadline detection in RT Mach. Currently, the deadline detector timer is bound to the periodic clock, the same clock that generates timer interrupts. Ideally, the deadline detector would utilize a separate clock, which would always be set to the next deadline of the active task following a context switch. Otherwise, a problem occurs if a deadline is between timer interrupts. Then a thread can continue to execute past this deadline, and the miss will go undetected as long as the thread completes before the next interrupt. If the deadline is immediately after a timer interrupt, then the task could potentially continue running for as much as a full timer interrupt interval. Thus, our experimental method suffers because we can potentially scale task sets beyond their actual breakdown utilization, and it will not be noticed on the target system. This problem is worsened for large timer interrupt intervals, where the potential for scaling error is greatest. Thus, the measured data point at 16 msecs for the avionics task set is significantly above the analytical point predicted by the model. The reader will note similar differences in the other figures.

Another cause for difference between the worst case performance assumed by the analytical model and the actual performance measured by the experiment is that the analytical model assumes worst case numbers for overhead and blocking terms. For the sufficient schedulability condition we must use worst case numbers, however, these worst cases may never occur, or occur to a far lesser extent, in the measured implementation. Thus, the measured task sets may scale higher than the sufficient bound bound if the worst case(s) do not occur. For example, the model assumes that the critical instant begins immediately after a timer interrupt, effectively delaying the response time by a full timer interrupt period and contributing a large blocking component. If a critical instant does not occur precisely after a timer interrupt, we would not maximize the blocking, and the measured breakdown utilization would be higher than expected. As before, with a large timer interrupt period, the blocking effect predicted by the analytical model is large, so the difference in blocking in the measured case is correspondingly larger.

Execution Time	Period
1180	2500
4280	40000
10280	62500
20280	1000000
100280	1000000
25000	1250000
Utilization	88.40%

Table 2: INS Task Set

The experiment was repeated for the Inertial Navigation System (INS) task set [18]. However, the highest priority task in INS, shown in Table 2, has a period of 2.5 msecs. This means that the timer interrupt interval must be less than 2.5 msecs for the task set to meet its deadlines. We again analytically and experimentally determined the breakdown utilization of the task set at timer interrupt interval of 1 and 2 msecs. The results are shown in Figure 5(b). The data point at 1 msec is below the predicted worst case bound. We believe this is because of the other activity on the target system that was not eliminated, such as network or other device interrupts. A number of these interrupts could easily render the task set unschedulable, especially because it has such small periods. In fact, the INS task set demonstrated a large variance in the measurements during the weekday, heavily loaded hours. The variance was less for the lightly loaded measurements. Additionally, if the interrupts or other system threads in the system required service from a large non-preemptable section of the kernel outside of the scheduler, the the schedulable utilization of the INS task set would be greatly diminished. Notice that the measured data point at 2 msecs was well above the predicted worst case, which again is explained by the limitations of the detection method. We note that INS was not schedulable with timer interrupt interval greater than 2.5 msecs, as was predicted by the model.

We next created a new task set, called ins_big, which scaled both the periods and execution times of the INS set by ten times. This was done to establish another set of data points similar to INS, but without the effect of the small periods. The results of the analytical and experimental scaling are shown in Figure 5(c). The difference between the ideal and burdened breakdown utilizations is small. This is because the periods of the task set are very large, and the relative cost of the operating system is small. As the timer interrupt period gets large, though, there is a bigger difference because of the detection mechanism.

Finally, we evaluated the costs of the operating system scheduler using a synthetic task set called test1.set. This task set was created to show the effect that the timer interrupt interval can have on blocking. The periods of the threads are multiples of the timer interrupt interval of 16 msecs. By offsetting the start time of every thread to be just after a timer interrupt, we try to maximize blocking for every job of every thread. Figure 5(d) shows the results of this experiment. The curve labeled Breakdown Utilization with Overhead Only (No Blocking) was plotted by analytically computing the breakdown utilization without accounting for blocking. The actual measured results lie between what was expected from a system without blocking and the fully burdened system. However, they track much better than the other task sets. We believe that the weakness of the detection method contributed to higher measured utilization.

Thus, the analytical model for the Real-Time Mach scheduler proved a successful predictor of the actual performance for the task set of real-time threads. However, the measured performance did vary from what was predicted, for several reasons. First of all, the analytical model provides only a sufficient condition, because of worst case blocking and overhead assumptions. The measured results can be expected to have higher schedulability if the worst case does not occur.

Secondly, the detection mechanism contributed to a large disparity in the measured results, especially for large timer interrupt intervals. This can be seen consistently from all figures. Finally, if there is other activity in the target system, such as network interrupts or interference from another device (disk, mouse), then other parts of the operating system will be invoked, which can introduce non-preemptable blocking segments. This will degrade utilization, as was illustrated well by the INS task set. This does point to an important facet of real-time scheduling theory. The scheduling model must account for every source of interference to provide a proper bound. Otherwise, timing guarantees can not truly be made. To verify our models better, we need to improve the detection mechanism on the target system and isolate it from the network and other interference. However, to make our scheduling models better we need to account for these effects as well.

6 Conclusions

In real-time computing, correctness depends not only on the results of computation but also the time at which outputs are produced. This paper strives to close the gap between the theoretic foundations of real-time systems design and the reality of implementation through an operating system. The first step is to develop scheduling models for operating system schedulers. Even the implementation of the scheduler introduces significant cost into the real-time system equation that must be accounted for to guarantee timing correctness.

In this paper we have developed a scheduling model for the Real-Time Mach operating system scheduler. We have developed a methodology for characterizing the components of the scheduler. We have characterized the RT Mach scheduler on the target platform using worst case scenarios for the different components. We then used that characterization to develop analytical models for how task sets should perform on the target system. We developed a method for measuring the performance of real-time tasks on the target system using synthetic threads with tightly controlled timing characteristics. Using these synthetic threads, we were able to verify experimentally the predicted performance of the analytical model.

We would like to improve the characterization method. The use of *pixie* to measure kernel code as a user program is effective, but difficult. Ideally, the measurements would be obtainable from a user program without specially modifying the kernel. However, the measurements differ from standard OS benchmarks, in that they first require the worst case scenarios to formed. This problem needs further study.

In addition to the improvements noted above, in both experimental method and in our modeling work, we are currently working to expand this theory in several directions. We have a preliminary framework for including aperiodic tasks (with aperiodic servers), syn-

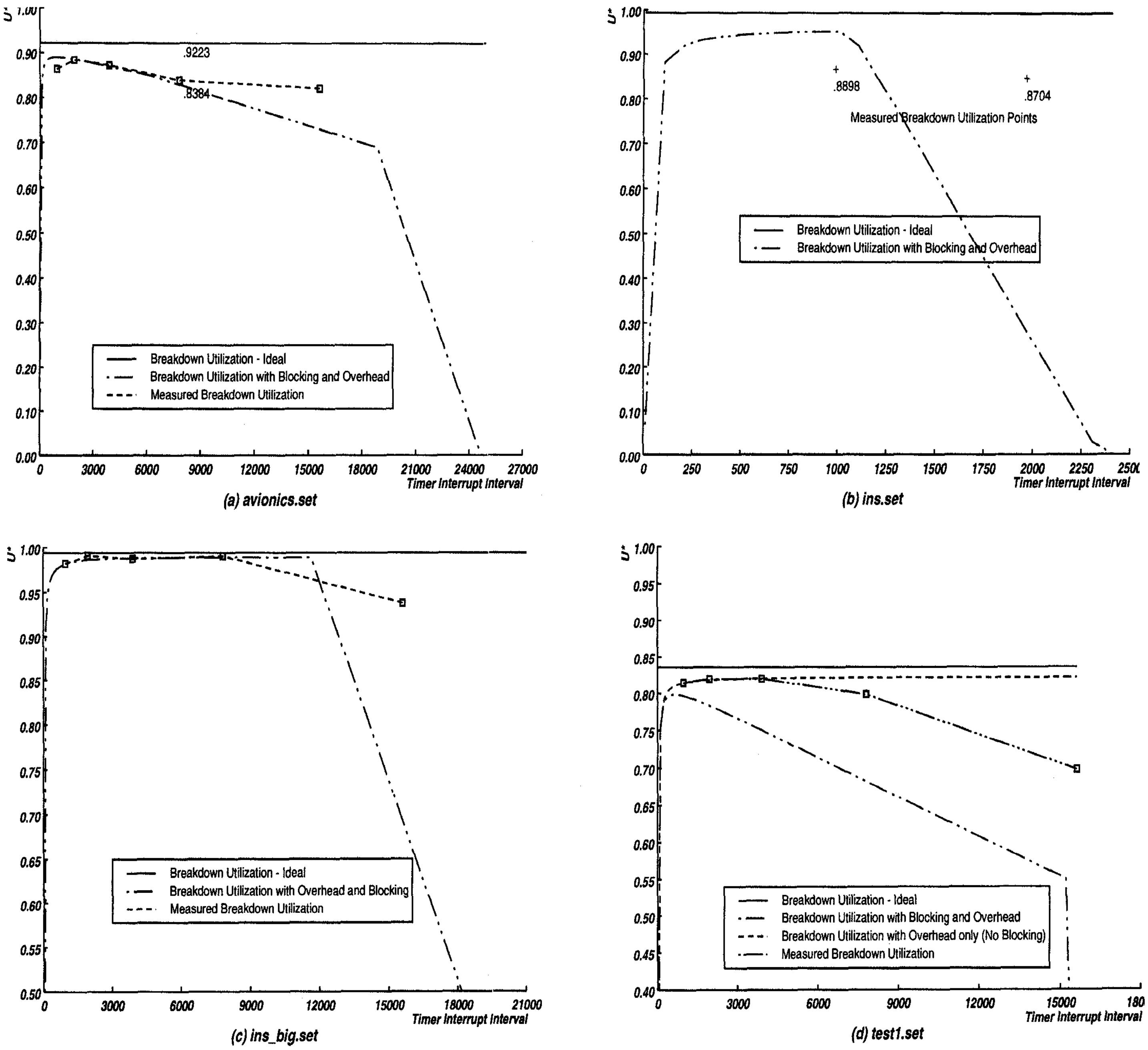

Figure 5: Analytical vs. Experimental Results

chronization, and IPC functions. The methods described here have also been helpful in the continual development of RT Mach, by allowing the designers the luxury of being able to quantitatively explore the OS design space.

References

[1] D. Katcher, H. Arakawa, and J. Strosnider, "Engineering and analysis of fixed priority schedulers," *To appear in IEEE Transactions on Software Engineering*, Accepted in 1993.

[2] D. Katcher, H. Arakawa, and J. Strosnider, "Engineering and analysis of real-time microkernels," *Proceedings of 9th IEEE Workshop on Real-Time Operating Systems and Software*, vol. 1, pp. 15–19, May 1992.

[3] K. Low, S. Acharya, M. Allen, E. Faught, D. Haenni, and C. Kalbfleisch, "Overview of real-time kernels at the superconducting super collider laboratory," *correspondence.*

[4] "Mips computer systems: Mips language programmer's guide," 1986.

[5] J. Strosnider, J. Lehoczky, and L. Sha, "The deferrable server algorithm," *To appear in IEEE Transactions on Computers*, 1993.

[6] B. Sprunt, *Aperiodic Task Scheduling for Real-Time Systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, August 1990.

[7] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, pp. 1175–1185, September 1990.

[8] J. Lehoczky and L. Sha, "Performance of real-time bus scheduling algorithms," *ACM Performance Evaluation Review, Special Issue*, vol. 14, May 1986.

[9] J. Stankovic, "Real-time computing systems: The next generation," *IEEE Tutorial on Hard Real Time Systems*, 1988.

[10] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," *IEEE Real Time Systems symposium*, 1989.

[11] M. Harbour, M. Klein, and J. Lehockzy, "Fixed priority scheduling of periodic tasks with varying execution priority," *Proceedings of the 1991 IEEE Real-Time Systems Symposium*, vol. 1, pp. 116–128, December 1991.

[12] D. Katcher, H. Arakawa, and J. Strosnider, "Engineering and analysis of fixed priority schedulers," *CMU/ECE Technical Report CMUCAD-91-10*, pp. 1–39, December 1991.

[13] H. Tokuda, T. Nakajima, and P. Rao, "Real-time mach: Towards a predictable real-time system," *Proceedings of USENIX Mach Workshop*, 1990.

[14] M. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young, "Mach: A new kernel foundation for unix development," *Proceedings of the Summer 1986 USENIX Conference*, pp. 93–113, July 1986.

[15] D. Black, D. Golub, D. Rashid, R. Dean, A. Forin, J. Barrera, H. Tokuda, H. Malan, and D. Bohman, "Microkernel operating system architectures and mach," *Journal of Information Processing*, December 1991.

[16] C. D. Locke, D. R. Vogel, and T. J. Mesler, "Building a predictable avionics platform in ada: A case study," *Proc. of Real-Time Systems Symposium*, pp. 181–189, December 1991.

[17] C. D. Locke, D. R. Vogel, L. Lucas, and J. B. Goodenough, "Generic avionics software specification," *Carnegie Mellon University Technical Report*, vol. CMU/SEI-90-TR-8, December 1990.

[18] M. W. Borger, "Vaxeln experimentation: Programming a real-time periodic task dispatcher using vaxeln ada 1.1.," *Technical Report, Software Engineering Institute, Carnegie Mellon University*, September 1987.