A. Tevanian, Jr. Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments. *Ph.D. Thesis*, Carnegie Mellon University, 1987.

H. Tokuda, T. Nakajima, P. Rao. Real-Time Mach: Towards Predictable Real-Time Systems. *Proceedings of the USENIX 1990 Mach Workshop*, October 1990.

D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. *Proceedings of the USENIX Summer Conference*, June 1990.

T. Hand. Real-Time systems need predictability. *Computer Design* RISC Supplement, August 1989.

J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

Intel Corporation. i960 Extended Architecture Programmer's Reference Manual. 1992.

M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, pp. 390-95, October 1986.

M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

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

D. Katcher, S. Sathaye, and J. Strosnider. Fixed Priority Scheduling with Limited Priority Levels. *IEEE Transactions on Computers*, 1994.

D. Katcher and J. K. Strosnider. Fixed Versus Dynamic Priority Scheduling, A Case Study. *CMU Technical Report*, August 1993.

J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. *IEEE Real Time Systems Symposium*, 1989.

C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, vol. 30, pp. 46-61, January 1973.

C. Locke, D. Vogel, and T. Mesler. Building a Predictable Avionics Platform in Ada: A Case Study. *Proceedings of the IEEE Real Time Systems Symposium*, 1991, pp 181-189.

R. Mraz. A RISC-Based Architecture for Real-Time Computation. *Ph.D. Thesis*, Carnegie Mellon University, 1992.

D. Nagle, R. Uhlig, T. Mudge and S. Sechrest. Optimal Allocation of On-chip Memory for Multiple API Operating Systems. *21st International Symposium on Computer Architecture*, 1994.

D. Niehaus, K. Ramamritham, J. Stankovic, G. Wallace, and C. Weems. The Spring Scheduling Co-Processor: Design, Use, and Performance. *Proceedings of the IEEE Real Time Systems Symposium*, 1993, pp 106-111.

J. K. Ousterhout. Why aren't Operating Systems Getting Faster as Fast as Hardware. *Proceedings of the Summer 1990 USENIX Conference*, pp. 247-256, June 1990.

G. Papadopoulos and D. Culler. Monsoon: An Explicit Token-Store Architecture. *17th Annual International Symposium on Computer Architecture*, 1990.

G. Papadopoulos and K. Traub. Multithreading: A Revisionist View of Dataflow Architectures. *18th Annual International Symposium on Computer Architecture*, 1991.

R. Rashid, et. al. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1987.

R. Rashid, et. al. Mach: A System Software Kernel. *Proceedings of the 34th Computer Society International Conference COMPCON 89*, February 1989.

6. Conclusion

Application performance has improved tremendously over the last few years. Unfortunately, operating system performance on identical platforms have not improved at a similar pace. This can be attributed to microprocessor architects focusing mainly on improving application performance and ignoring operating system performance. Some recent work has addressed this gap, but only examined average-case performance issues.

This paper explored another approach to closing this performance gap that can offer greater performance benefits and also can be directly useful for real-time systems requiring specific deadline guarantees. Specifically, we explored whether hardware-support for micro-kernel primitives can improve the performance of an operating system. The paper focused on the Intel 80960XA Microprocessor's hardware-assisted operating system primitives. We found an average operating system primitives performance improvement factor of 3. We also qualitatively explained why the hardware-assisted version realizes such gains. We introduced three real-time task sets and quantitatively illustrated the benefits of a hardware-assisted implementation. The hardware implementation not only allowed the task sets to be schedulable over larger timer intervals, but also, at certain timer intervals, was closer to the ideal system than to the software-only implementation.

This work can be extended with the use of a processor simulator where the architecture can be arbitrarily varied. The approach delineated in this paper can then be used to ascertain the ideal hardware/ software boundary for operating system code.

Acknowledgments

The Authors would like to thank Roy Reed of Intel Corporation for helping us understand the Intel 80960XA Processor in depth. The Authors are also indebted to Dan Katcher of Carnegie Mellon University for his assistance with reviewing earlier drafts of this paper. Finally, the Authors wish to thank the anonymous reviewers for suggesting improvements to the paper.

References

A. Agarwal, J. Hennessy, and M. Horowitz. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Transactions on Computer Systems*, November 1988, pp. 393-431.

T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The Interaction of Architecture and Operating System Design. *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 108 - 120, April 8, 1991.

H. Arakawa, D. Katcher, J. Strosnider, and H. Tokuda. Modeling and Validation of the Real-Time Mach Scheduler. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993.

B. Bershad. The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems. *Proceedings of the 1992 USENIX Workshops on Microkernels*, 1992.

S. Chatterjee and J. Strosnider. Issues in Hardware Support for Micro-kernel Based Operating Systems. *CMUCSC-94-1 Technical Report*, Carnegie Mellon University, March 1994.

J. Chen and B. Bershad. The Impact of Operating System Structure on Memory System Performance. *14th Symposium on Operating System Principles*, 1993.

D. Clark. and J. Emer. Performance of the VAX-11/780 Translation Buffer: Simulation and Measurements. *ACM Transactions on Computer Systems*, Feb 1985.

Thread Type	Priority	Execution Time (μsec)	Period (μsec)	Deadline (μsec)
Voice	1	1175	6000	6000
MIDI	2	9.4	12000	12000
JPEG (36 Hz) at 0.5 bpp	3	1880	27000	27000
JPEG (36 Hz) at 0.5 bpp	4	1880	33000	33000
File Transfer	5	5000	1000000	1000000

Table 4: Characterization of a Multimedia Task Suite

Thread Type	Priority	Latency Speed-up
Voice	1	1.19x
MIDI	2	1.27x
JPEG (36 Hz) at 0.5 bpp	3	1.18x
JPEG (36 Hz) at 0.5 bpp	4	1.16x
File Transfer	5	1.28x

Table 5: Latency Speed-up for Multimedia Tasks using Hardware Support

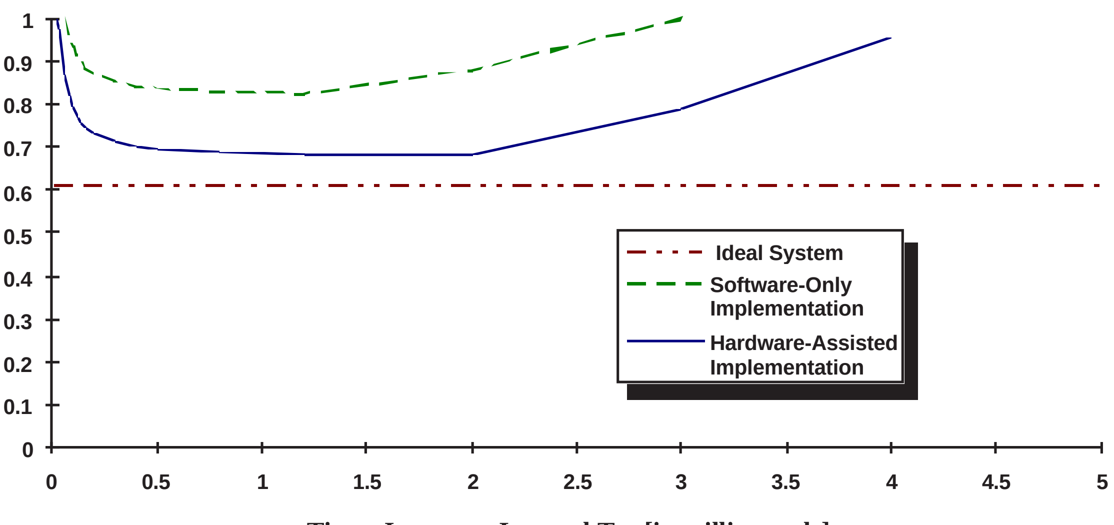
where α is the scaling factor that is applied uniformly to all the run times in a task set. Specifically, *U'* is the utilization at which all deadlines are met, but any further increase in will cause one or more tasks to miss their deadlines (Arakawa93).

Our validation was done using a synthetic workload with the attributes of the avionics application suite described earlier on the hardware-assisted version of Real-Time Mach executing on the Intel 80960XA Processor. To minimize measurement errors, we replaced the traditional heartbeat timer interrupt mechanism with an event-driven timer for the Real-Time Mach scheduler (Chatterjee94). The heartbeat timer implementation produces errors due to the granularity of detection points when executing the deadline detection algorithm (Arakawa93). An event-driven timer implementation eliminates this error.

As shown in Table 6, the predicted and measured breakdown utilization for the avionics task set were approximately equal. The predicted utilization was more pessimistic, as expected, by approximately 1%. The ideal breakdown utilization is approximately 91% for the avionics task set.

	U'
Ideal	0.91
Event-Driven, Predicted	0.83
Event-Driven, Measured	0.84

Avionics Task Set
Table 6: Utilization for the Hardware-Assisted Version of Real-Time Mach



Timer Interrupt Interval T_{tic} [in milliseconds]

Figure 10: **Maximum Schedulable Saturation (S_{max})**

milliseconds for the hardware-assisted and software-only implementations, respectively. Smaller intervals result in excess overhead whereas larger intervals increase blocking. The software-only implementation is unable to schedule the modified INS task set for timer intervals below 80 microseconds and above 2 milliseconds, whereas the hardware-assisted implementation is successful over a wider range, for timer intervals between 40 microseconds and 4 milliseconds. For a 2 millisecond timer interrupt interval, the hardware-assisted implementation has a minimum overhead burden of 0.07 as compared to a minimum burden of 0.27 for the software-only implementation. The resulting overhead burden reduction factor is approximately 4.1. Alternatively, the hardware-assisted implementation is capable of scheduling roughly 20% additional high priority load and still guarantee all task deadlines.

We further analyze the impact of hardware-support for a multi-media task suite (Table 4) (Katcher94). Priorities were assigned using the rate-monotonic algorithm. The multimedia task contains a high-priority thread with period of 6 msecs. Therefore, the timer interrupt interval had to be decreased to schedule the thread. The software-only version has a Degree of Schedulable Saturation of 0.74 and 0.86 at timer tick intervals of 1 ms and 2 msec, respectively. On the other hand, the hardware-assisted version has Degree of Schedulable Saturation of 0.52 and 0.62 for timer tick intervals of 1 and 2 ms, respectively. As a result, the hardware-assisted version of Real-Time Mach decreases the Degree of Schedulable Saturation by approximately 23% over the software-only version. Furthermore, thread latency improves by 21% on average for the multi-media task, as shown in Table 5. Thread latency is defined as the ideal execution time plus the thread's overhead and blocking times.

5.2 Avionics Task Set Validation

Analytic modeling of a system provides a clean and simple method of predicting user and system performance. However, it is imperative to check that the predicted results agree with empirical data. For validation we use the breakdown utilization metric, U' (Lehockzy89). It is found by uniformly scaling all thread execution times, C_i, while holding the periods fixed, until the task set is just schedulable. It is defined as

$$U' = \sum_{i=1}^{n} \frac{\alpha C_i}{T_i},$$

schedule without missing any thread deadlines. If $S_{max} \leq 1$, then the task set is schedulable. A smaller Degree of Schedulable Saturation indicates that more threads may be scheduled without missing any deadlines.

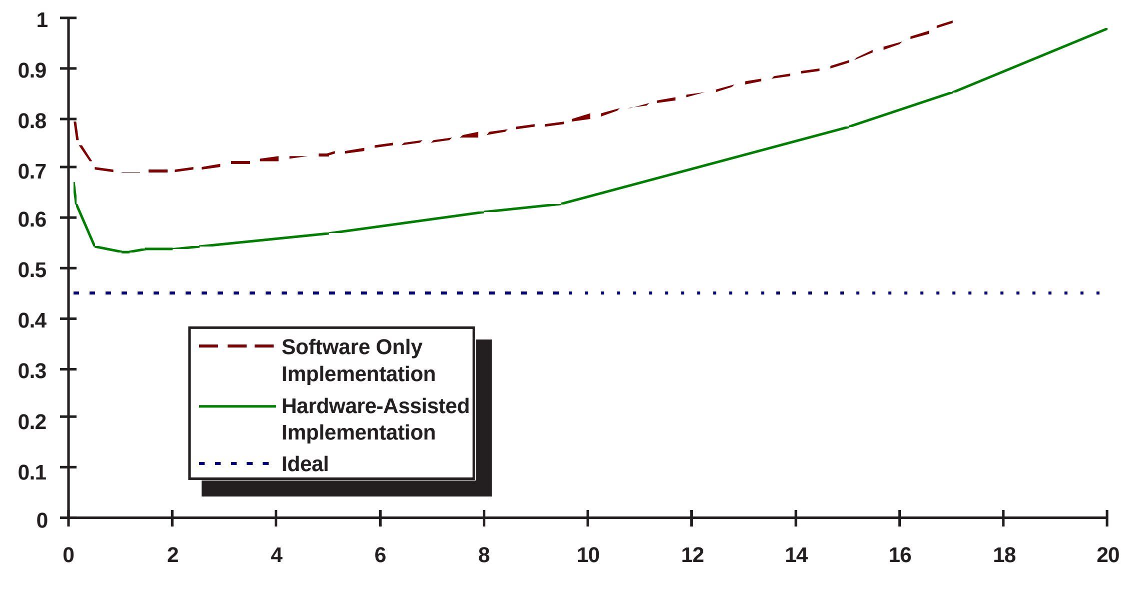
We first consider the avionics application (Locke91) which is representative of an avionics control system. It consists of fifteen real-time threads, with periods ranging from 25 milliseconds to 1 second. Figure 9 plots the Degree of Schedulable Saturation as a function of the timer interrupt interval, T_{tic}, for ideal, software and hardware-assisted versions of Real Time Mach. Since the timer interrupt interval is user-controlled, it is important to understand the impact of hardware-assisted scheduling for a range of T_{tic} values.

The ideal line uses Equation 5-2 but where $Overhead_j$, $Overhead_{sys}$, and $Blocking_i$ are all zero. For the other two curves, as can be seen from Equation 5-2, small values of T_{tic} translate to excessive system overhead, whereas large values translate into high blocking costs.

As expected, the ideal system has the lowest Degree of Schedulable Saturation; the ideal plot corresponds to zero operating system burden. The difference between the ideal and the other plots represents the *operating system burden* required to support the avionics task set. The hardware-assisted implementation enjoys a significant performance advantage across the entire range of T_{tic} values. The operating system burden is 9% for the hardware-assisted implementation and 24% for the software-only implementation at the most efficient timer intervals. The operating system burden *reduction factor* for the hardware-assisted implementation over the software-only implementation is the ratio of the two burdens or 2.7. Another interpretation would be that the hardware-assisted implementation can support roughly 15% additional high priority loads and still meet all deadlines.

The second real-time application that we evaluated was an Inertial Navigation System used on Navy ships. This system is characterized by six real-time threads ranging in frequency from six milliseconds to 1.25 seconds. Figure 10 plots the Degree of Schedulable Saturation as a function the timer interrupt interval, T_{tic}, for this application. Again both the software and hardware-assisted implementation are compared with the ideal.

The timer interrupt intervals yielding the lowest Degree of Schedulable Saturation are 2 and 1.2



Timer Interrupt Interval T_{tic} [in milliseconds]

Figure 9: **Maximum Schedulable Saturation (S_{max}) for the Avionics Task Set**

4.2 Generalization

The measured speed-up in Table 3 compares the execution of the software-only and hardware-assisted versions of Real-Time Mach, both implemented on the 80960XA platform. As stated above, this method eliminates auxiliary variables such as cache speed, memory latency and compiler technology differences between various microprocessor platforms. However, the software-only version is still constrained by the additional 80960XA state required to execute the microcoded instructions for the hardware-assisted version.

We calculated another performance improvement factor, labeled "Adjusted Speed-Up" in Table 3, that reflects the expected performance improvement factor after eliminating the 80960XA artifacts from the software-only version of RT Mach. For the software-only version, we eliminated execution times to save and resume a 80960XA process during a context switch and execution time to check the dispatch port after an interrupt. Note that the changes did not significantly affect the scheduler ratio whereas it lowered thread load and save state ratios slightly. We believe that this adjusted improvement factor is what we would have measured had we executed the software-only version on a generic scalar RISC platform with identical cache size, memory latency and clock speed as the 80960XA.

Reduced cache and TLB miss rates, reduced register saving, and increased parallelism contribute to the performance advantage of the hardware-assisted scheduler. For example, the software-only $C_{non\text{-}preempt}$ symbolically requires two "pseudo-context switches": one to switch from a thread to the scheduler and another to switch back. Due to bad operating system locality (Agarwal88, Clark85), initial code execution after a context switch results in significant number of cache and TLB misses. The hardware-assisted scheduler executes in ROM and has no such constraints (Hennessy90). It also accesses internal registers, resulting in less external register saves and restores.

The hardware-assisted scheduler exploits machine parallelism not available at the macro-instruction level on scalar machines. Specifically, using microcode, the processor can issue instructions to all available functional units simultaneously. Note that this advantage remains for superscalar processors. For example, moving from a two-instruction issues per cycle to a four-instruction issue requires doubling the hardware for a 20-25% improvement in performance (Johnson91). Most of this extra hardware is required to check register/memory dependencies between instructions. However, using microcode, assuming that all dependencies are resolved after compilation, the processor can issue the maximum possible number of instructions per cycle without requiring the extra hardware.

5. Impact of OS/Architecture on Real-Time Threads

In this section, we evaluate the comparative operating system burden for the two Real-Time Mach implementations each running two representative real-time applications. For the software-assisted version, we used the costs derived for the "Adjusted Speed-Up" column in Table 3.

5.1 Task Set Modeling

To quantitatively analyze the relative thread schedulability for a particular task set, we use the *Degree of Schedulable Saturation* (Katcher94), S_{max}, defined as

$$S_{max} = max_{1 \le i \le n} S_i, \tag{5-1}$$

where S_i is from Equation 2-1:

$$min_{0 < t \le D_i} \; S_i(t) = \sum_{j=1}^{i} \frac{C_j + Overhead_j}{t} \left\lceil \frac{t}{T_j} \right\rceil + \frac{Overhead_{system}}{t} + \frac{Blocking_i}{t} \tag{5-2}$$

The Degree of Schedulable Saturation indicates the maximum amount of additional load that a system may

2. These components map directly to the 80960XA hardware-supported operating system primitives. Table 3 lists the benefits as performance ratios of hardware-assisted to software-only implementations for *n* threads for these primitives. The first column lists the measured ratios and the second lists the adjusted ratios after the execution times of 80960XA artifacts (to be explained in Section 4.2) were deleted from the software-only version.

OS Components	**Measured Speed-up**	**Adjusted Speed-up**
C_{sched}	$\frac{5.45 + 0.13n}{0.1n + 1}$	$\frac{5.45 + 0.13n}{0.1n + 1}$
C_{load}	2.45	2.21
C_{store}	1.31	1.18

Ratio of Software-Only to Hardware-Assisted Execution Time
Table 3: Performance Improvement Factor for *n* Threads

The execution time of the hardware scheduler, C_{sched}, is a function of the number of threads because manipulating the start and run queues require *O(n)* searches. The software-only scheduler spends a slightly larger time searching the start and run queues than the hardware-assisted scheduler. The other three components correspond to individual thread scheduling and are not functions of the number of threads in the system. These are listed in Table 3 under “Measured Speed-Up.”

Maintaining a non-preemptable kernel for the hardware-assisted version, as described in Section 3.2, introduces some additional costs. For any kernel execution time, C_{kernel}, that involves a trap to the kernel, the new execution time, $C_{non-preemptable\ kernel}$, is as follows:

$$C_{non-preemptable\ \ kernel} = C_{set-priority} + C_{kernel} + C_{restore-priority} \quad (4\text{-}1)$$

where

$C_{set-priority}$	time to change to highest priority
$C_{restore-priority}$	time to restore original thread priority

As stated in Section 2, a non-preemptable kernel induces C_{system}. Arakawa, Katcher, Strosnider and Tokuda (1993) map this component to C_{exit}. As a result, for the hardware-assisted implementation, C_{system}, is as follows:

$$C_{system} = C_{set-priority} + C_{exit} + C_{restore-priority} \quad (4\text{-}2)$$

This additional cost only affects kernel calls and not the components listed in Table 3. However, since the blocking cost, C_{system}, is due to such calls, this variable increases slightly for the hardware-assisted version. This increase is reflected in the data for Section 5 but does not affect any of the variables in this section.

In summary, the hardware-assisted version requires slight modifications to the operating system to guarantee functional correctness. This additional software adversely affects overhead and blocking costs. However, the benefits of a hardware-assisted scheduler dwarf these disadvantages. On average, the hardware-assisted scheduler reduces the time spent execution operating system code by a factor of 3.

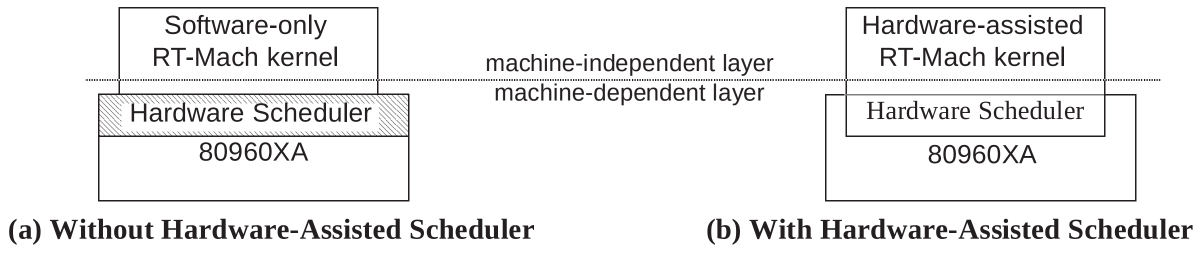


Figure 7: **Two implementations of Real-Time Mach on the Intel 80960 Processor.**

between the two implementations is approximately two, versus five for the scheduler only, the aggregate ratio falls. Performance ratio for C_{exit} is approximately constant because both versions implement the start queue search routines in software and have approximately equal execution times.

Thread switch performance ratios slowly decrease as the number of threads increase because of similar reasons to $C_{non-preempt}$. Execution times for intra- and inter-task thread switches are equal for the hardware-assisted version because the 80960XA does not understand the intra-task thread abstraction. As a result, even though the address space pointer for the preempted and preempting threads are identical, the scheduler executes code required to switch the virtual address space.

Thread switch execution times are a function of the number of threads in the system because enqueueing a thread on the dispatch port requires a search of the start queue. As stated in Section 3, for the hardware-assisted implementation, moving a thread from the start queue to the run queue does not require a search of the dispatch port. Furthermore, the hardware-assisted implementation updates the start queue when the scheduler moves a thread to the dispatch port. This execution time is a function of the number of threads in the system. Since this search occurs after a thread exit for the software-only implementation, we include this time in C_{exit}. To maintain similarities with the software-only implementation, we also include the corresponding execution time to search the start queue for the hardware-assisted implementation in C_{exit}.

The above components can be further sub-divided into finer components, as explained in Section

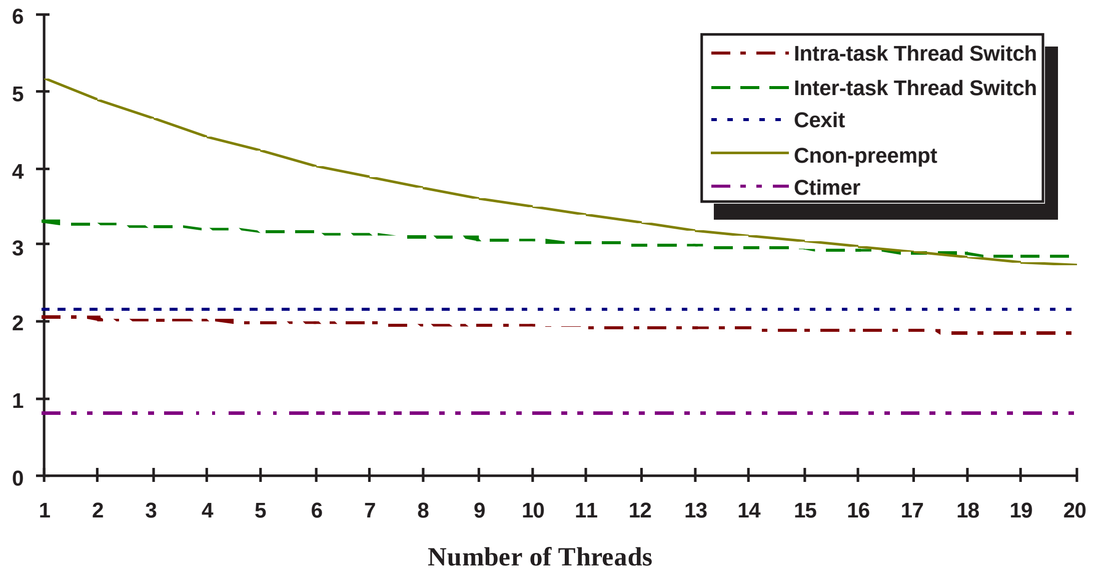


Figure 8: **Performance Improvement Factor using the 80960XA Scheduler**

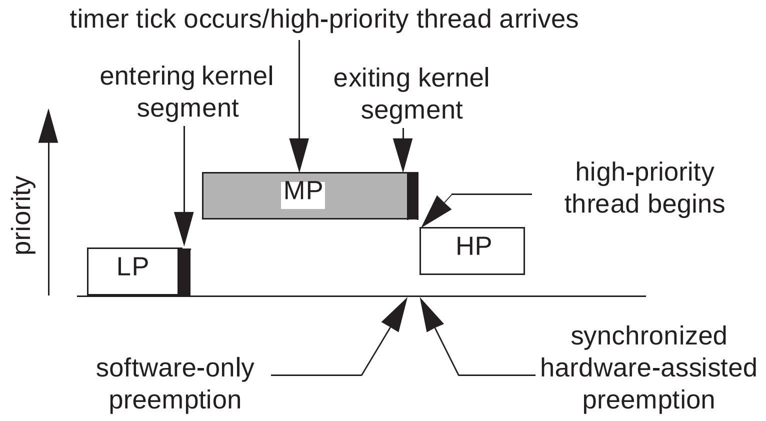

timer tick occurs/high-priority thread arrives
entering kernel segment
exiting kernel segment
priority
MP
high-priority thread begins
HP
LP
synchronized hardware-assisted preemption
software-only preemption

Figure 6: **Synchronized Hardware-Assisted Scheduler and Real-Time Mach Kernel**

in a critical section (in kernel segment), a higher-priority thread (HP) arrives at the timer tick. Because the hardware-assisted scheduler is not aware that a non-preemptable section is being executed (shaded area), the scheduler will attempt a context switch. The desired (correct) point for a context switch occurs after the non-preemptable kernel segment has finished executing.

To prevent early thread preemption, the 80960XA hardware scheduler must synchronize with the RT Mach kernel. One solution (Chatterjee94) actually increases blocking for the hardware implementation by a small amount over the software-only RT Mach implementation. This algorithm temporarily increases the priority of the current thread to priority thirty-one whenever a user invokes a non-preemptable kernel segment. All user threads, by definition, execute at lower priorities. As shown in Figure 6, this prevents any untimely thread switches by the hardware scheduler because a lower priority thread cannot preempt a higher priority thread. This algorithm reduces the maximum number of thread priorities to thirty. Another solution provides lesser average-case blocking than the previous solution but degrades worst-case blocking (Chatterjee94). Since real-time scheduling theory is based on worst-case analysis, we use the former solution in our analysis in Section 4.

4. Operating System Primitives Performance Evaluation

4.1 Performance Numbers

This section evaluates the relative performance of operating system primitives for the software-only and hardware-assisted versions of Real-Time Mach. The primary figure of merit used is the *performance improvement factor*, defined as the ratio of the time to execute the software-only implementation over the time to execute the hardware-assisted implementation.

Both implementations of Real-Time Mach are executed on the 80960XA platform, as shown in Figure 7. This approach eliminates the auxiliary variables, such as cache size, bus width and speed, memory latency and compiler technology, when contrasting the performance of the software-only versus the hardware-assisted implementations on separate machines. Start and stop instructions were placed inside the functions being measured. We subtract the execution time of all timer instructions for both versions and execution of all 80960XA-specific artifacts (process and semaphore objects from Figure 1) from the software-only version when comparing performance.

Figure 8 plots the performance improvement factors for the hardware-assisted over the software-only implementation of Real-Time Mach for n $(1 \le n \le 20)$ threads. The execution time ratio of the two scheduler implementations is slightly larger than five for $C_{non-preempt}$ with no threads. As the number of threads in the system increases, the time to search the start queue increases. Since the ratio of search time

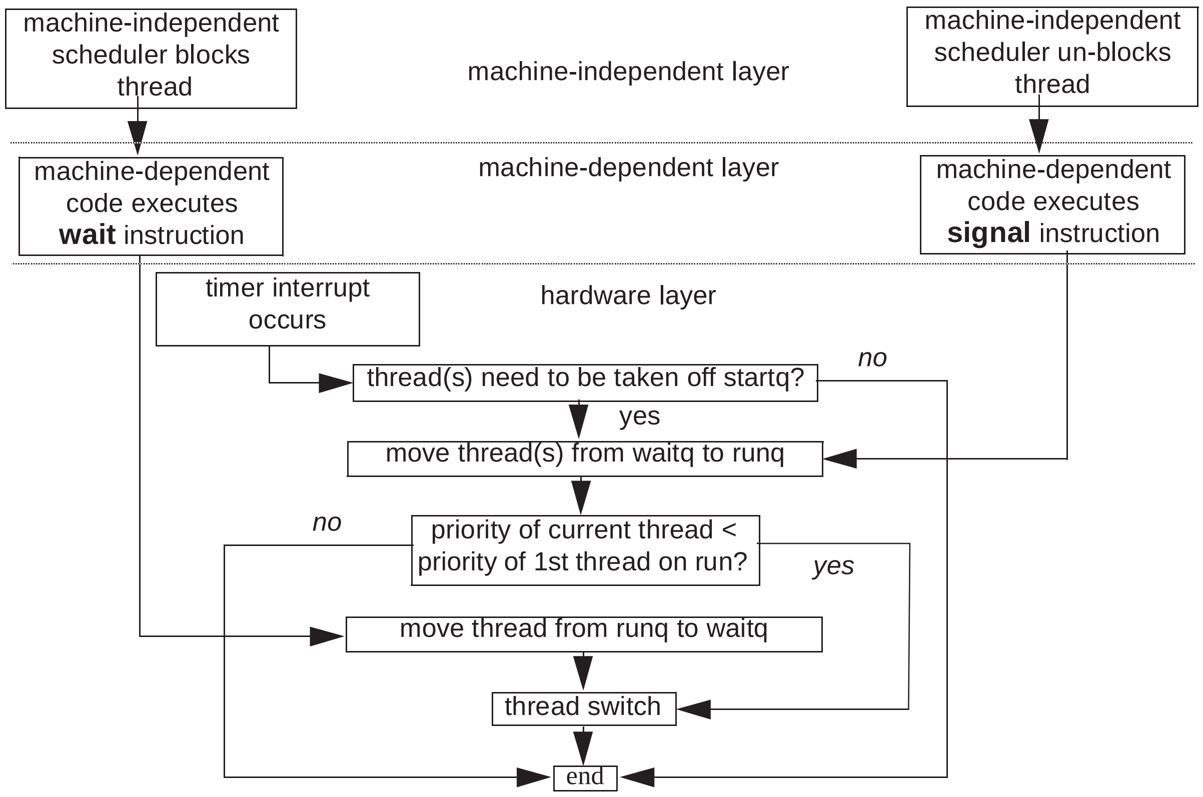

machine-independent scheduler blocks thread
machine-independent layer
machine-independent scheduler un-blocks thread
machine-dependent code executes **wait** instruction
machine-dependent layer
machine-dependent code executes **signal** instruction
timer interrupt occurs
hardware layer
thread(s) need to be taken off startq?
no
yes
move thread(s) from waitq to runq
no
priority of current thread < priority of 1st thread on run?
yes
move thread from runq to waitq
thread switch
end

Figure 4: **Flowchart of scheduling requirements at each layer**

the 80960XA accomplishes this functionality. *Process* objects maintain the complete state of RT Mach threads, including their global registers and frame pointer.

3.2 Non-preemptable Kernel Segments: C_{system}

The non-preemptable design of the RT Mach kernel induces blocking (C_{system} in Equation 2-4). For example, a higher priority thread may be ready to execute when the processor is performing a non-preemptable RT Mach kernel routine on behalf of a lower priority thread. On the software-only RT Mach implementation, blocking occurs until that segment completes and a thread switch occurs. On the other hand, the hardware-assisted implementation may autonomously switch threads while the processor is executing a non-preemptable kernel segment. Since the RT Mach kernel is non-preemptable, this may lead to an undefined kernel state. This can be seen in Figure 5. While the low-priority thread (LP) is executing

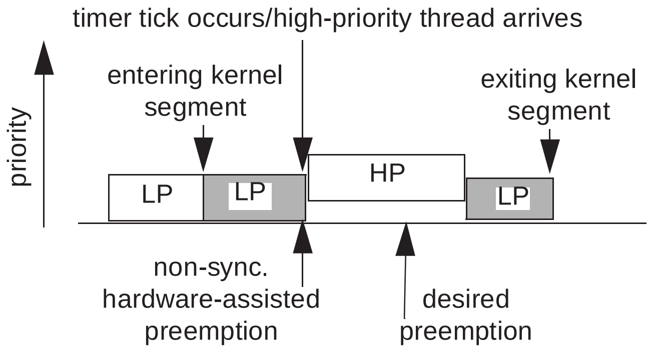

timer tick occurs/high-priority thread arrives
entering kernel segment
exiting kernel segment
priority
LP
LP
HP
LP
non-sync. hardware-assisted preemption
desired preemption

Figure 5: **Non-synchronized Hardware-Assisted Scheduler and Real-Time Mach Kernel.**

3.1 OS Scheduling: C_{sched}, C_{load} and C_{store}

This section looks at the two implementations of Real-Time Mach from the perspective of scheduling threads and loading and saving thread states. The job of a fixed-priority scheduler is to always execute the highest priority thread eligible for execution. A currently running thread may be preempted by a higher priority thread that becomes eligible for execution. A thread may also block voluntarily, *e.g.*, waiting for a semaphore.

Fixed-priority scheduling is implemented in the software-only version of Real-Time Mach using two software-maintained queues (Table 2). The *run queue* contains threads, in priority order, that are eligible for execution. A vacant run queue signifies that the system contains no eligible threads corresponding to the run queue's priority. The *start* queue contains threads, in start time order, waiting to become eligible for execution. The fixed-priority scheduler always executes the first thread in the highest-priority, non-vacant run queue. At timer interrupts, the scheduler executes, checks the start queue for eligible threads and moves threads from the start to the run queue if required. If the thread at the head of the run queue has a higher priority than the currently executing thread's priority, the scheduler preempts the current thread, places it at the rear of the run queue, changes address space, and runs the thread at the head of the run queue.

Threads exit by trapping to the RT Mach scheduler. The scheduler kills aperiodic threads at this time. On the other hand, the scheduler moves periodic threads to the start queue to await its next start time. The scheduler then executes the next highest-priority thread in the run queue.

The hardware-assisted version of Real-Time Mach implements fixed-priority scheduling using a singular *dispatch* queue. The 80960XA dispatch queue is identical to RT Mach's run queue. The hardware-assisted version of RT Mach schedules a thread completely by sending it to the dispatch queue. The thread is then enqueued on one of 32 lists, based on its priority. When the processor is executing a priority i thread and a preempting thread of priority $i + 1$ or greater arrives at the dispatch queue, the preempting thread seizes the processor.

Unlike the software-only RT Mach implementation, where the RT Mach scheduler can execute periodic threads, the 80960XA scheduler can only schedule *aperiodic* threads. Hence, the hardware-assisted implementation requires software to allow periodic thread scheduling. Note that the difference between aperiodic and periodic thread handling is only at the thread exit stage. The added complexity of this exiting routine, however, is minimal because of the use of 80960XA *semaphore* objects to control thread execution.

Unlike the software-only implementation, thread exit for the hardware-assisted implementation does not require a trap from the user to the kernel. Instead, the thread executes the **wait** instruction, automatically suspending the thread. The software-only implementation also updates the next start time for the exiting thread before moving the thread to the start queue. Since the hardware-assisted implementation eliminates scheduler invocation when a thread exits, the scheduler updates the next start time for the exiting thread prior to dispatching the thread via the **signal** instruction. The execution times of these two update routines remain equal.

Semaphore objects are also utilized by the high-level Real-Time Mach scheduler to control the low-level scheduler for the hardware-assisted version. The exact functionality and control mechanisms of the hardware scheduler is detailed in Figure 4. The issues involved in implementing the hybrid scheduler for the hardware-assisted version are detailed in (Chatterjee94).

Before executing a preempting thread, the states of the preempted and preempting threads must be saved and loaded, respectively. For the software-only version, software routines in Real-Time Mach provide this functionality. Specifically, the thread state of the preempted thread is saved in its machine-dependent process control block and the thread state of the preempting thread is recovered from its own machine-dependent process control block. On the other hand, for the hardware-assisted version of Real-Time Mach,

3. Analysis of Hardware Support

The previous section identified the fundamental scheduler components required in any operating system. This section analyzes whether hardware-support for operating system primitives can reduce the execution time for these fundamental primitives: C_{sched}, C_{system}, C_{load}, and C_{store}.

A *generic* RISC processor provides minimal hardware support for operating system functions. It does provide essential features, such as faults and interrupts, that software cannot implement without minimal hardware support. However, it does not provide hardware support for thread switching, thread scheduling, loading and saving thread state, or trapping from a user thread to the kernel. This has a direct impact on the components in the real-time scheduling equations. The Intel 80960XA Microprocessor, on the other hand, provides extensive hardware support for operating system functions. Its architectural features include a fixed-priority hardware scheduler, and *process* and *semaphore* objects abstractions. Table 1 summarizes the differences.

Generic RISC Processor	Intel 80960XA Microprocessor
minimal hardware support for OS	extensive hardware support for OS
fault and interrupt support provided	fault and interrupt support provided
no support for: • thread scheduling • thread switching • loading and saving thread state	OS hardware support: • fixed-priority hardware scheduler • process objects • semaphore objects

Table 1: Comparison of Operating System Hardware Support Differences between a Generic RISC Microprocessor and the Intel 80960XA Microprocessor.

The remainder of this section describes two implementations of Real-Time Mach, one using only generic RISC hardware features and another using the unique hardware features of the 80960XA. The *software-only* version of Real-Time Mach is representative of any generic RISC architecture. This implementation exclusively executes machine instructions found on any RISC microprocessor. The *hardware-assisted* version of Real-Time Mach substitutes 80960XA-specific hardware-assisted instructions for a series of generic RISC instructions, as shown in Table 2. Sections 4 and 5 will then quantitatively examine the impact of hardware support on operating system and application performance.

Software-Only RT-Mach Scheduler	Hardware-Assisted RT-Mach Scheduler
software maintains two queues: • run: threads eligible for execution • start: threads waiting for next period	hardware maintains singular queue: • dispatch: threads eligible for execution • software maintains start queue
software maintains thread state	hardware *process* object maintains thread state

Table 2: Implementation Differences between the Software-only and Hardware-assisted Versions of Real-Time Mach

Equation 2-1 sums the execution and overhead times of all equal- and higher-priority threads and blocking due to lower-priority tasks and divides by t to normalize the equation. For a non-ideal system, a context switch manifests itself as $Overhead_j$, timer tick as $Overhead_{system}$, and the non-preemptability of a kernel segment as $Blocking_i$ (Figure 3). Blocking is defined as the time after a high-priority thread becomes eligible for execution but before it actually begins execution.

Arakawa, Katcher, Strosnider and Tokuda (1993) characterized the Real-Time Mach scheduler using the above framework, substituting specific worst-case overhead and blocking terms:

$$Overhead_j = C_{preempt} + C_{exit} \tag{2-2}$$

$$Overhead_{system} = C_{timer}\left\lceil \frac{t}{T_{tic}} \right\rceil \tag{2-3}$$

$$Blocking_i = T_{tic} + C_{system} + \sum_{j=i+1}^{n} C_{non-preempt}\left\lceil \frac{t}{T_j} \right\rceil \tag{2-4}$$

where

C_{timer}	Execution time of the timer interrupt routine,
$C_{preempt}$	Scheduler execution time including a thread switch,
$C_{non-preempt}$	Scheduler execution time excluding a thread switch (occurs when scheduler runs but preemption of thread not required),
C_{exit}	Execution time for a thread to exit and save its state,
T_{tic}	Time between timer interrupts,
C_{system}	Execution time of longest non-preemptable kernel routine.

The above variables can be mapped to more fundamental components:

$$C_{preempt} = C_{sched} + C_{load} + C_{store} \tag{2-5}$$

$$C_{non-preempt} = C_{sched} \tag{2-6}$$

$$C_{exit} = C_{sched} + C_{trap} + C_{store} \tag{2-7}$$

where

C_{sched}	Execution time of the scheduler,
C_{load}	Execution time to load the state of the preempting thread,
C_{store}	Execution time to save the state of the preempted thread,
C_{trap}	Execution time to trap from the user to the kernel.

In summary, a non-preemptable kernel implementation and the timer interrupt interval induce blocking, denoted $Blocking_i$. The execution time of the timer interrupt routine creates system overhead, $Overhead_{system}$, and thread switch routines manifest in thread overhead, $Overhead_j$. Section 3 next examines how hardware can be used to alleviate the execution time of some of these operating system primitives, some possible pitfalls in using hardware-support and their solutions.

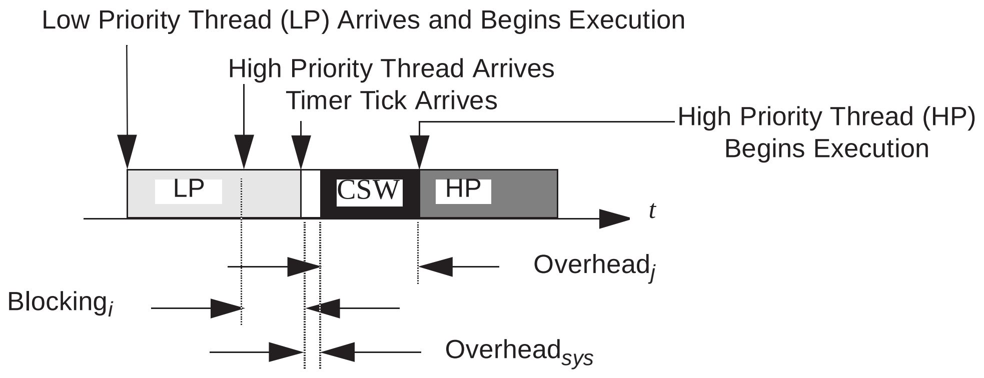

Low Priority Thread (LP) Arrives and Begins Execution
High Priority Thread Arrives
Timer Tick Arrives
High Priority Thread (HP) Begins Execution
LP
CSW
HP
t
Overhead$_j$
Blocking$_i$
Overhead$_{sys}$

Figure 3: **Illustration of Overhead and Blocking**

2. Modeling Real-Time Operating Systems

A real-time system is required to provide correct outputs within certain time constraints. Historically, timing correctness was achieved via hand-crafted timelines. This approach lead to brittle systems which were difficult to extend, upgrade and maintain. The modern approach is to use preemptive, priority driven scheduling algorithms which dynamically bind resource allocation at run-time via priorities. *Earliest deadline* and the *rate monotonic* scheduling algorithms are two of the most popular preemptive, priority-driven scheduling algorithms for real-time applications. They were shown to be the optimal dynamic and fixed priority scheduling algorithms, respectively, by Liu and Layland (1973). Although the analysis developed in this paper is fixed priority specific, the methodology has also been applied to the earliest deadline first based operating systems by Katcher and Strosnider (1993).

Liu and Layland provided only a least upper schedulability bound for fixed priority scheduling. Joseph and Pandya (1986) and Lehoczky, Sha and Ding (1989) both explored an exact (necessary and sufficient) schedulability criterion. These papers modeled ideal operating systems with perfect preemptability, i.e., no blocking, and zero overhead. Katcher, Arakawa and Strosnider (1992) extended the Lehoczky, Sha and Ding scheduling theoretic framework to include the implementation costs of real operating systems running on real processors. Their operating system scheduling model provides the framework used in this paper to evaluate the performance gains associated with hardware support for operating system functions.

The generic model for fixed-priority periodic scheduling on a dedicated processor, *i.e.*, no network I/O, may be characterized as follows. Given a real-time task set composed of *n* threads τ_i, $0 < i \leq n$, with periods T_i, deadlines D_i $(D_i \leq T_i)$, and worst-case execution times, C_i, the task set is schedulable for fixed priority scheduling if the following condition is true:

$$\forall i,\ 0 < i \leq n,\ \ min_{0 < t \leq D_i} \sum_{j=1}^{i} \frac{C_j + Overhead_j}{t} \left\lceil \frac{t}{T_j} \right\rceil + \frac{Overhead_{system}}{t} + \frac{Blocking_i}{t} \leq 1 \qquad (2\text{-}1)$$

where

$Overhead_j$	Kernel execution time performing a service on behalf of a user thread,
$Blocking_i$	Execution time during which a higher priority thread cannot execute,
$Overhead_{system}$	System overhead not attributed to any one thread.

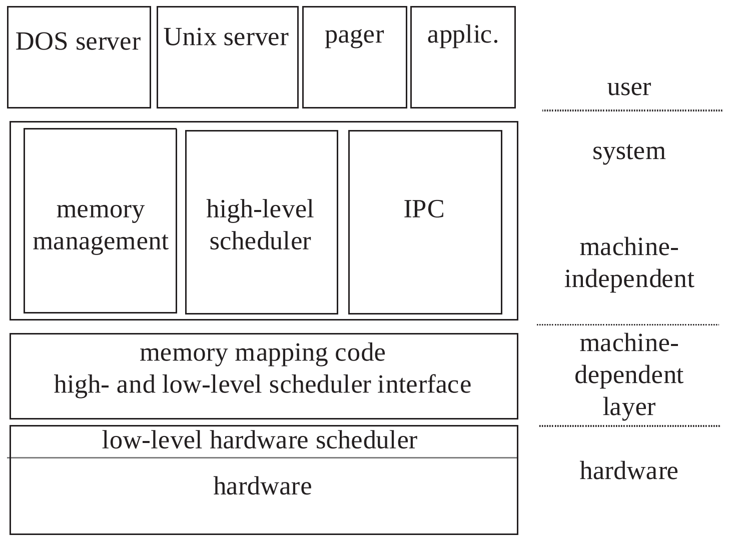


Figure 2: **Modular design of operating system with an hardware scheduler**

scheduling due to voluntary thread blocking or blocking due to IPC, memory or device I/O constraints, are operating system-specific and should be implemented in software to retain maximum flexibility.

Therefore, our approach is to execute IPC, memory management and *high-level* scheduling in software. An hardware scheduler will alleviate the cost of *low-level* scheduling. If a thread blocks voluntarily or because of IPC, memory or device constraints, the high-level scheduler will instruct the low-level scheduler to block the thread and to schedule another thread. The low-level scheduler will do the actual thread preemption, queue management and thread state save and load. Likewise, when required, the high-level scheduler must instruct the low-level scheduler to re-schedule the thread.

This paper compares two implementations of Real-Time Mach (Tokuda90) on the Intel 80960XA Microprocessor (Intel92), one using common architectural features found on most RISC machines and another using additional hardware found on the 80960XA. The 80960XA, which directly supports basic operating system primitives in hardware, provides an excellent platform to quantitatively explore whether hardware-support can improve the performance of operating systems. Real-Time Mach, a super-set of Mach 3.0 (Rashid89), is a widely used representative micro-kernel based operating system.

We use a formal operating system scheduling model to quantitatively evaluate the relative performance of the operating system primitives, and to determine the relative operating system burden of the two approaches on three representative real-time applications. The results on the relative performance of the operating system primitives are general and can be readily extended to the non-real-time application domain. The results on the relative operating system burden are specific to the real-time/multimedia processing domain characterized by time-constrained processing. Although results are provided for three real-time applications, the methodology can be readily applied to arbitrary real-time applications.

Section 2 provides an introduction to real-time scheduling theory and illustrates how to model Real-Time Mach using real-time scheduling theory. Section 3 looks at Real-Time Mach scheduler functions in more detail and states how 80960XA hardware-support can improve performance. Section 4 quantitatively compares the performance of these functions with and without hardware support. Section 5 analyzes the corresponding impact of this hardware support for several real-time tasks. Section 6 summarizes the work and explores future directions.

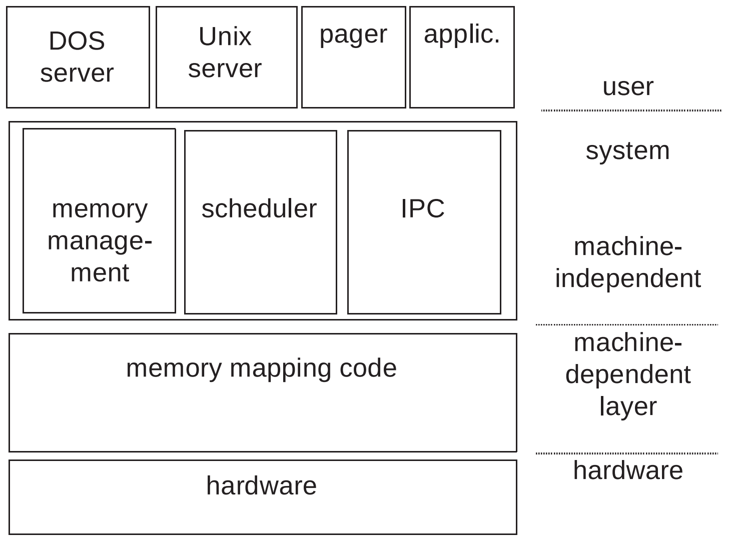


Figure 1: **The layered-approach to micro-kernel design.**

pipeline must support dual-instruction streams but the incremental hardware addition over a singular-instruction stream is minimal. This approach has the potential to improve thread performance by 20% on a scalar processor with an embedded kernel (Mraz92). However, this embedded kernel provides no mechanisms for higher-level operating system functionality, such as memory and file management, inter-process communications or multiprocessor support, required for today's complex applications.

Another possible solution is to customize the cache configuration and size for a given operating system and application load (Nagle94). However, memory hierarchies improve average-case performance but have no effect on worst-case performance. Because real-time systems are concerned about worst-case behavior, this solution cannot be used. Second, because predictability is important for real-time systems (Hand89), memory hierarchies, due to collisions between operating system and user data in caches (Agarwal88, Chen94), exacerbate the problem. Separate caches for operating systems and applications do not rectify this problem (Agarwal88). Furthermore, since cache hit rates are in the high ninety percent range (Hennessy90), further increasing cache size or changing cache configuration will only result in marginal performance improvements. As a result, for real-time systems, other approaches to improving performance must be investigated.

Our approach is to execute *common* operating system primitives in hardware, thereby reducing system and user cache conflicts, *i.e.*, increasing predictability, and significantly improving the performance of those primitives. This approach is particularly beneficial for current modular micro-kernels that only execute a core set of primitives frequently. This paper specifically analyzes the viability of a hardware-assisted fixed-priority scheduler. To allow maximum flexibility, all other operating system functions will be executed in software. We identify key components of operating system scheduling, detail methods of hardware-support, and illustrate performance analysis between software and hardware-assisted fixed-priority schedulers from the perspective of both operating systems and real-time applications.

1.2 Methodology

This paper divides scheduling functions into two groups. *Low-level* scheduling functions, such as queue management and thread context switching, are fundamental to all operating systems. As a result, a simple but extremely fast scheduler could be implemented in hardware and still be general enough for use with most micro-kernel based operating systems. On the other hand, *high-level* scheduling, defined as

Accepted to the *Journal of Real-Time Systems*, 1994.

Quantitative Analysis of Hardware Support for Real-Time Operating Systems

Saurav Chatterjee and Jay Strosnider
Department of Electrical & Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract Microprocessor architects, supported by advances in VLSI technology, have been enormously successful at steadily accelerating the performance of application software. However, operating system performance has lagged due to a divergence between operating system and architectural trends. Unfortunately, some recent work in this area has targeted average-case performance improvements with little or no consideration for the worst-case behavior that must be considered for real-time applications. This paper explores whether one can improve the worst-case performance of operating systems, and as a result, the schedulability of real-time task-sets, using specific hardware-assisted operating system primitives without sacrificing flexibility. The Intel 80960XA Microprocessor, which directly supports basic operating system primitives in hardware, provides an excellent platform to explore operating system hardware and software boundary issues. This paper specifically analyzes the viability of an *hardware-assisted fixed-priority scheduler*. Using the Real-Time Mach operating system, we did two ports to the 80960XA: one representative of generic RISC implementations, and another which exploited the hardware-supported operating system primitives. We measured the performance of the operating system primitives in both cases and found an average performance improvement factor of 3 for the hardware accelerated version. We applied a formal scheduling *model* to evaluate the relative performance of the two implementations for two representative real-time applications. The hardware accelerated version reduced operating system burden by factors of 2.66 and 4.1 for the avionics and inertial navigational system task sets, respectively.

1. Introduction

1.1 Motivation

Operating system and architecture paths have diverged over the last few years. As a result, application performance has improved at a faster rate than operating system performance (Ousterhout90). This is because current architectural support for operating systems assumes a monolithic kernel design, while current operating system trends are towards modular, layered micro-kernel structures (Anderson91). File management, Unix system support and other high-level functionality traditionally present in monolithic operating systems are relegated to user-level server threads for micro-kernel structures (Golub90). This user-level server mechanism increases flexibility by allowing multiple operating system servers to concurrently execute on the micro-kernel. The micro-kernel provides basic operating system functionality such as thread scheduling, memory management and inter-process communications. This functionality is implemented using a layered structure, as shown in Figure 1. This approach also improves portability to numerous architectural platforms by requiring only the machine-dependent code to be hardware-specific. Unfortunately, compared to monolithic kernels, this modularity degrades the performance of the operating system (Chen94).

One previous method to increase system performance via hardware focused on adding a co-processor for static and on-line scheduling (Niehaus93). Another similar approach subsumed operating system costs in application pipeline stalls (Mraz92). This approach, inspired by multiple instruction-stream dataflow machines (Papadopoulos90, Papadopoulos91), embeds the scheduler kernel in hardware on a single processor and uses pipeline stall cycles during thread execution to execute scheduler code. The