Hardware Support for Real-time Operating Systems

Paul Kohout[1]
EVI Technology , LLC.
7138 Columbia Gateway Dr.
Columbia Maryland 21046
pkohout@evitechnology.com

Brinda Ganesh and Bruce Jacob
Department of Electrical and Computer Engineering
University of Maryland, College Park
Maryland 20742
{brinda,blj}@eng.umd.edu

[1] Work done when Paul was at UMD.

ABSTRACT

The growing complexity of embedded applications and pressure on time-to-market has resulted in the increasing use of embedded real-time operating systems. Unfortunately, RTOSes can introduce a significant performance degradation. This paper presents the Real-Time Task Manager (RTM)—a processor extension that minimizes the performance drawbacks associated with RTOSes. The RTM accomplishes this by supporting, in hardware, a few of the common RTOS operations that are performance bottlenecks: task scheduling, time management, and event management. By exploiting the inherent parallelism of these operations, the RTM completes them in constant time, thereby significantly reducing RTOS overhead. It decreases both the processor time used by the RTOS and the maximum response time by an order of magnitude.

Categories and Subject Descriptors

A.0 [General]: Conference Proceedings

General Terms

Performance, Design, Experimentation.

Keywords

RTOS, Hardware-Software Codesign.

1. MOTIVATION

RTOSes have become extremely important to the development of real-time systems as reflected by the growing market for RTOSes; half a billion dollars in shipments of RTOSes will be sold in 2002 [5]. RTOSes provide services for better hardware abstraction multitasking, task synchronization etc. However, the benefits that RTOSes provide do not come for free. The use of RTOSes has several important effects on various performance parameters of real-time systems.

Processor Utilization: The fraction of processing power that an application is able to use. In order to provide services like multitasking, preemption, and numerous others, the RTOS introduces an overhead which lowers the processing power available to the application.

Response Time: The time it takes for a real-time system to response to external stimuli. This delay is highly dependent on several factors, including whether or not the RTOS is preemptive and whether polling or interrupts are used to sense the stimulus. The effects of RTOSes on response time vary widely, but, in general, RTOSes increase response time to varying degrees.

Much of the performance degradation caused by RTOSes can be traced to their core operations, namely task scheduling, time management, and event management. Analysis of these functions reveals that they are executed frequently, that they perform inter-related actions, and that these actions exhibit parallelism. That parallelism cannot be exploited by software-based implementations; however, hardware-based solutions can exploit this parallelism and lower the performance degradation. Also, the rapidly dropping cost of logic [7] has made it possible to put custom hardware for enhancing frequently executing operations on embedded processors, without increasing costs significantly.

These factors suggest that RTOSes would greatly benefit from a custom hardware solution. This is the motivation for the Real-Time Task Manager (RTM), a memory-mapped on-chip peripheral designed to optimize task scheduling, time management, and event management that is compatible with a wide range of RTOSes. Measurements have been taken of the performance impact of the RTM on models of realistic real time systems. These measurements show that processor utilization and maximum response time are both reduced by an order of magnitude.

2. BACKGROUND

To better understand that the commonly seen bottlenecks in RTOSes, we present some of the details regarding their nature — what they are, how they are implemented and how often they are invoked.

Task Scheduling: Process of determining which task should be running at any given time. The most commonly implemented approach to scheduling in commercial RTOSes is based on task priorities [8]. Priority-driven schedulers assign each task a priority and execute the task with the highest priority that is ready.

Time Management: The RTOS gets its sense of time by a periodic interrupt generated every clock tick by a hardware timer. The rate of the generation of this clock tick is the system clock frequency which is not to be confused with the CPU clock frequency. Typically the system clock frequency is on

CODES-ISSS'03, October 1-3, 2003, Newport Beach, California, USA.

Table 1. RTOS Function Details

	Implementation / Overhead	Performed When
Scheduling	• Unsorted Ready List - Task Selection has Linear Overhead • Sorted Ready List -Task Selection has Constant small overhead, Task Insertion on Ready Queue has Linear Overhead • Bit-Vectors[6] - Constant Overhead implementation for systems with unique and static priorities.	• System calls which change task status e.g. Priority Change. Scales with workload size. • Preemptive System- After Timer Interrupt. Scales with system clock frequency. • Non Preemptive System - Task Completion Points, After Timer Interrupt if System is Idling. Scales with workload size.
Time Management	• Each task has individual delay counter containing its absolute delay. Timer Update requires updating each counter and has a linear overhead • UNIX Callout Table[2]. Delay counters hold delays relative to that of previous element. Updates have variable overhead. Task Insertions has linear overhead.	• Preemptive System - After Timer Interrupt. Scales linearly with system clock frequency. • Non Preemptive System - Task Completion Points, After Timer Interrupt if System is Idling. Scales with workload size.
Event Management	Similar to Task Scheduling with each event having its own Blocked Queue.	• Tasks use IPC. Dependent on extent to which workload uses IPC.

the order of kHz while the CPU clock is on the order of MHz. *Time management* refers to the RTOS's ability to allow tasks to use this mechanism to be scheduled at specific times i.e. have the tasks block for a specific time before becoming ready.

Event Management: Most real-time operating systems provide services for communication and synchronization between tasks—known as *interprocess communication (IPC)*. Examples are semaphores, message queues etc. Event management involves keeping track of which tasks are blocked i.e. waiting for IPC and which tasks should be released i.e. have to receive IPC on resource availability.

Note that as shown in Table 1, the overhead of the RTOS due to these operations is proportional to the product of frequency and complexity. Since both of these components may increase linearly with the number of tasks, there may be a quadratic relationship between the number of tasks in the system and the RTOS overhead due to them. In addition the overhead of task scheduling and time management increases linearly with the system clock frequency.

The longest time that it takes to perform any critical section of code (including task scheduling, event or time management) adds to the maximum response time in preemptive systems. However, in non-preemptive systems where interrupts are polled, a response task is delayed more by executing workloads than by RTOS operations. This is because workloads execute for longer periods than RTOS operations and thus disable interrupt response for longer periods.

3. RELATED WORK

Dick, Lakshminarayana, Raghunathan, and Jha first published a study of the power consumption of RTOSes in embedded systems [4] on an instruction-level simulator of the Fujitsu SPARClite processor, using embedded applications running on μC/OS. They suggested ways in which to design application software so that the power consumption is minimized.

Adomat, Furunäs, Lindh, and Stärner described the Real-Time Unit (RTU)—an external hardware module designed to perform RTOS functions [1]. The RTM-based RTOS lies at an intermediate point in the design spectrum whose two extremities are represented by a purely software based RTOS and the RTU — an exclusively hardware-based design. The RTU improves performance, but it does not allow for existing RTOSes to easily take advantage of its offerings.

Mooney and his co-authors [12,13] have explored hardware implementations of lock synchronization, deadlock detection, dynamic memory management in a multi processor system. These schemes effectively solve problems seen in a multi-processor system namely atomic access of data, potential resource conflicts. Our scheme is orthogonal to their scheme because it explores hardware acceleration for basic RTOS operations. In addition Mooney et al [14] have looked at building a cyclic scheduler in hardware. Unlike our design it is application specific and does not investigate support for alternate scheduling mechanisms like EDF etc.

4. REAL-TIME TASK MANAGER

The RTM is a hardware module that implements a task database and thus supports the key RTOS operations: task scheduling, time management, and event management. It is an on-chip peripheral that communicates with the core via a memory-mapped interface.

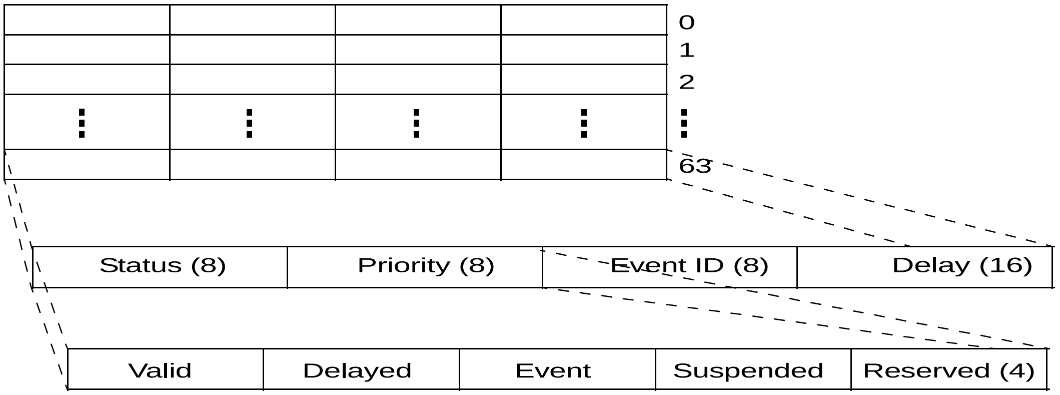

Figure 1. Reference RTM Data Structure

The reference RTM architecture has 64 records. The status, priority and event ID fields are each 8 bits wide and the delay field is 16 bits wide. Each record uses 40 bits of storage but is mapped into the I/O space at even-word (64 bit) boundaries to simplify and speed up addressing.

The RTM is pictured in Figure 1; where each record contains information about a single task. The RTM is accessed through the global address space and each record can be read from or written to as if it were any other array of structures.

A record is composed of four individually addressable fields. Each record contains a *status field*, containing several bits that describe the status of the corresponding record. The *valid bit* is necessary to indicate if that the record is used by a task. The *delayed bit* indicates that the task is waiting for the amount of clock ticks specified by the *delay field* before being ready-to-run. The *event bit* indicates that the task is pending on the event with the identifier specified by the *event ID field*. The *suspended bit* indicates that the task has been suspended. If the valid bit is set, but the delayed, event, and suspended bits are clear, then the task is ready-to-run. Finally, the *priority field* indicates the task's priority. Also, the maximum number of records is some fixed constant, such as 64 or 256. The RTOS can issue instructions to or retrieve certain information from the RTM via a set of memory mapped registers.

The RTM can be queried for the highest priority ready task and thus supports static-priority scheduling. The RTM does time management by decrementing the delay fields of all records by the number of clock ticks specified by the RTOS and updating the delay bit if this value reaches zero. Event management is

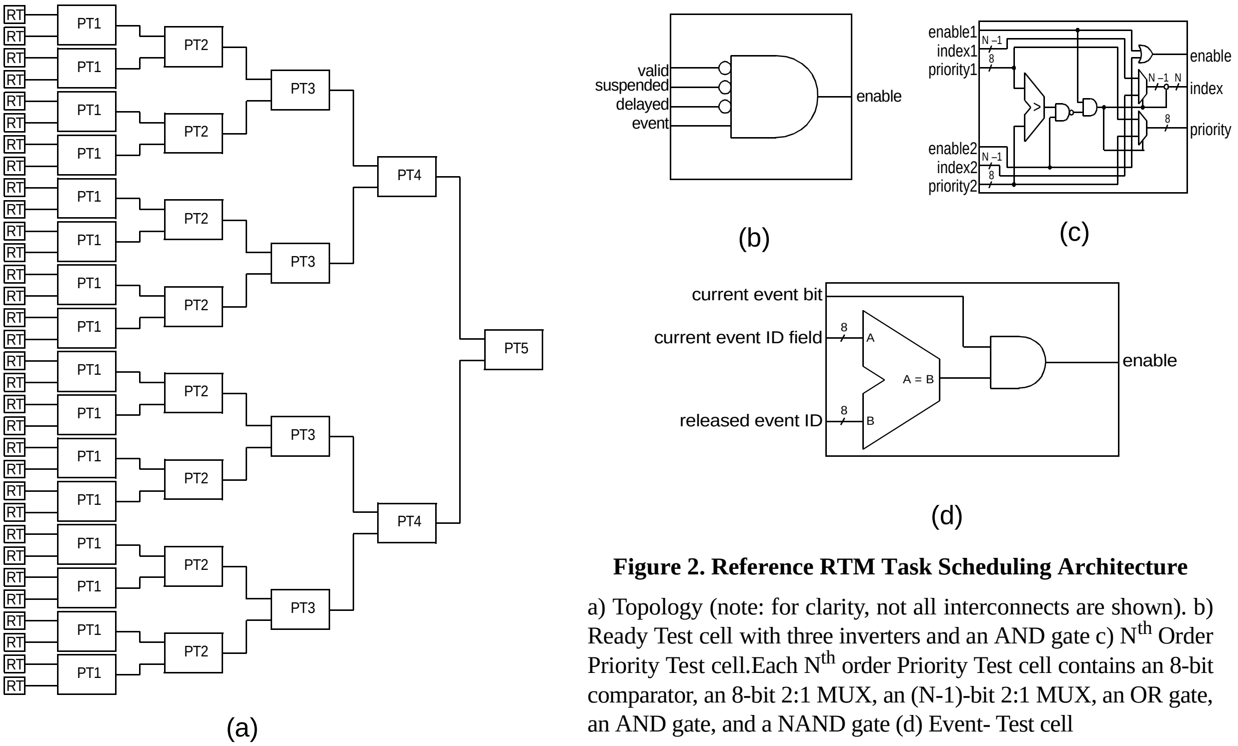

Figure 2. Reference RTM Task Scheduling Architecture

a) Topology (note: for clarity, not all interconnects are shown). b) Ready Test cell with three inverters and an AND gate c) Nth Order Priority Test cell.Each Nth order Priority Test cell contains an 8-bit comparator, an 8-bit 2:1 MUX, an (N-1)-bit 2:1 MUX, an OR gate, an AND gate, and a NAND gate (d) Event- Test cell very similar to static-priority scheduling. The RTM queries for the highest priority task that is pending on a given event identifier, instead of querying its data structure for the highest priority ready task.

5. ARCHITECTURE

A reference architecture is presented here to illustrate the complexity and scalability of implementing the hardware architecture of the RTM. For a 64 record RTM with entries as shown in the figure the total number of flip-flops required is 2304, which is small enough to easily be implemented. These parameters are based on common limits and are sufficient for the majority of real-time applications.

Task scheduling is performed using 64 Ready Test cells and a binary tree of 63 Priority Test cells. Owing to space considerations we show this for a 32 entry case in Figure 2. The Ready Test cells simply determine which tasks are ready-to-run. Each Priority Test cell takes in as input two task priorities and ready bits. It then outputs which of these is the highest priority ready task. These priority values propagate down the comparator tree until the single sixth order cell where the index of the overall highest priority ready task is output. Each N^{th} order Priority Test cell contains an 8-bit comparator, an 8-bit 2:1 MUX, an (N-1)-bit 2:1 MUX, an OR gate, an AND gate, and a NAND gate. The logic required task scheduling scales linearly with the number of records in the RTM (n RTM records requires n RT cells plus n-1 PT cells). The computational delay is proportional to the logarithm of the number of records. For a reasonable number of records i.e. the numbers typically used in production RTOSes, this implementation of task scheduling will be sufficiently fast and small.

In order to implement time management, 64 delay decrement cells are all that is required, as shown in Figure 3. Delay decrement cells consists of a simplified 16-bit adder and an AND gate. Delay cells decrement the delay field and perform a comparison to clear the delay bit when necessary. In this implementation, the logic scales linearly with the number of records; however, the computational delay is a constant. In designs where the CPU clock frequency is lower than the system clock frequency, the delay decrement cells can be reused and thus lower the number of delay cells required.

The event management is almost exactly the same as task scheduling. The only difference is that instead of Ready Test cells, it has Event Test cells, as seen in Figure 2. This allows the binary tree of Priority Test cells to be used for both scheduling and event management. The Event Test cells just check if the event bit is set and if the event ID of the resource being released matches the event ID in a specific record. They each use an 8-bit comparator and an AND gate. The combinational logic required for event management scales linearly, except that the majority can be shared with the task scheduler. Also, the computational delay is O(log(n)), as it is with scheduling.

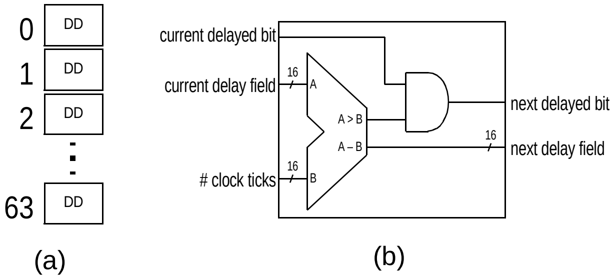

Figure 3. Reference RTM Event and Time Management Architecture

a) Topology (note: for clarity, not all interconnects are shown). b) Delay Decrement Cell.

The die area that the RTM needs is important in determining its feasibility. Based on existing area models for register files and caches [9], the RTM reference architecture requires approximately 2600 register-bit equivalents (RBEs). This is roughly equivalent to the die area used by a 32-bit by 64-word register file making it feasible to implement the RTM in hardware. This number is large compared to a simple core but is small compared to the whole chip. This is because a typical embedded processor consists of the core and other on-chip devices like memory and various I/O devices.

6. EXPERIMENTS

In order to formally justify the use of the Real-Time Task Manager in actual real-time systems, an accurate quantification of the effects that it has on performance is done by analyzing models of real-time systems that use RTOSes. All measurements are made on a inhouse cycle-accurate C-based simulator of the Texas Instruments TMS320C6201. The

Table 2. Details of RTOSes used in study

	µC/OS-II	**NOS**
Preemptive	Yes	No
IPC?	Yes	No
Task Scheduling	Constant -Bit Vector	Linear - Sorted Ready Queue
Time Management	Linear - Unsorted Queue	Linear - UNIX Callout Queue
Event Management	Constant - Bit Vector	n/a

The table gives details of the RTOSes used in the study. The details regarding the overhead of the implementation of the RTOS functions (scheduling, time management and event management) are given along with the method employed.

processor is a high-performance 32-bit VLIW fixed-point DSP with a 200 MHz clock that can issue eight 32-bit instructions per cycle [11]. The simulator is capable of loading the same executable binary files that run in actual systems and executing them exactly as they would on a real processor.

Two differing RTOSes have been used in this study: µC/OS-II, a popular commercial preemptive RTOS [6]; and NOS, a "homegrown" non-preemptive RTOS, that is representative of over 25% of RTOSes which are merely subsets of complete RTOSes [5]. More details are given in table below.

Several applications from the MediaBench suite [7] are used in this analysis including workloads like GSM decompress, ADPCM encode and decode and the encryption and decryption benchmark Pegwit. The benchmarks are modified so that they are separated into two periodic tasks, an input processing task and an output task, with periods of 20 ms. In µC/OS, semaphores are used to synchronize the access of the data channel by the two tasks. An additional noise task (with a period of 32 ms) purpose is to insert the type of background noise commonly present in real-time applications, e.g., many systems have LCD displays which have to be periodically refreshed. Aperiodic tasks which respond to external stimuli, such as I/O events are simulated as a polling task in NOS or by interrupts in µC/OS systems. The I/O events arrival times are geometrically distributed with an average arrival rate of 10 milliseconds.

7. RESULTS

The goal of the Real-Time Task Manager is to reduce the performance loss problem associated with RTOSes. In order to validate the success of the RTM, we characterize its effects on processor utilization, response time.

7.1 RTOS Overhead

For purposes of brevity we present the average processor utilization across 5 benchmarks for different workloads. For all benchmarks the performance was reasonably close to the average and for no benchmark did the configurations with the RTM perform worse than those without.

µC/OS-II: The RTOS processor utilization has been divided into five categories for system configurations that use µC/OS. These five categories include Scheduling, Interprocess Communication i.e. Event Management, Timer Interrupts, Processing Clock Ticks i.e. Time Management or updating the timer queue and a Miscellaneous category which includes task Creation etc.

As seen in the graph in fig. 4, the RTOS processor utilization without the RTM can be quite significant. The bulk of the reduction comes from converting the overhead of time management operations that ready tasks that reach their release times at every clock tick from a linear one to a constant one. The gains in event management and scheduling are lower, 14% and 30% respectively, because of the effectiveness of the software-based bit vector scheme employed in µC/OS. The timer interrupt overhead and miscellaneous category which are not optimized account for a constant less than 1% of the processor utilization, both with and without the RTM. The basic RTOS operations that the RTM implements result in the processing overhead required by µC/OS to be reduced by 60% to 90%.

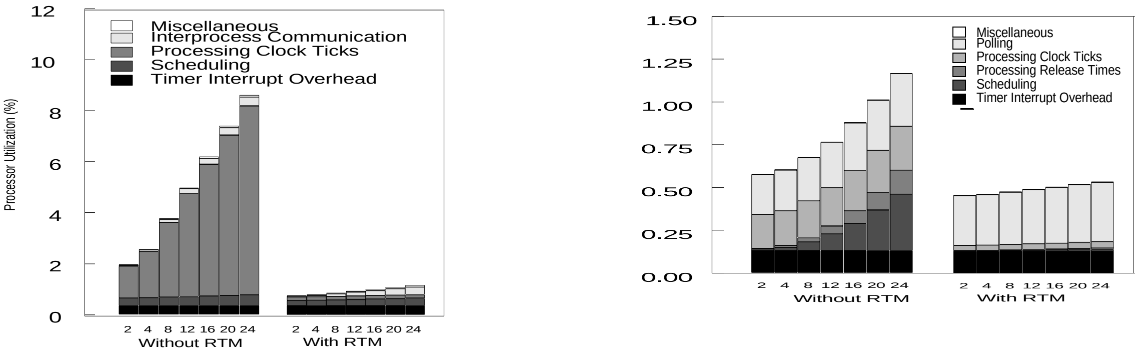

Table 3. µC/OS-II

Function	% Reduction
Scheduling	31
Processing Time	83 (average)
Event Management	14

Table 3. NOS

Function	% Reduction
Scheduling	98
Processing Time	85 (average)
Event Management	14
Polling	Increase 25%
Release Times	95%

Figure 4. Processor Utilization Using µC/OS-II and NOS

The percent of the total processing time spent executing the operations in each of these categories and how they vary with system load, for every benchmark tested, both with and without using the RTM. System load refers to the amount of processing power used by the application, which, in this case, is determined by the number of data channels processed in the application

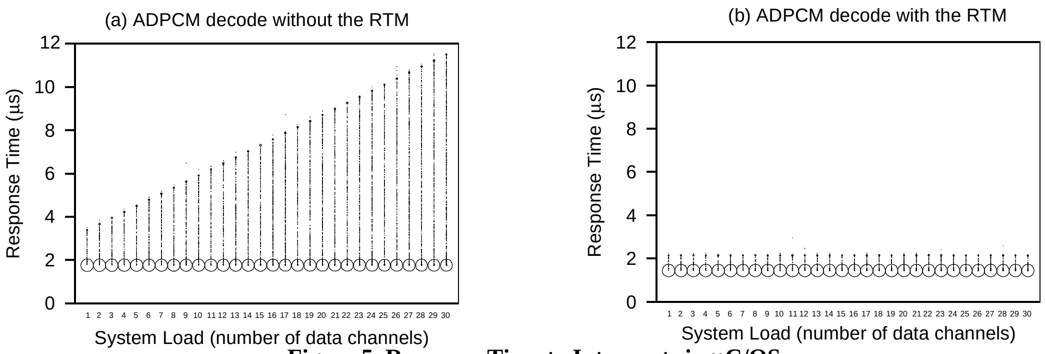

Figure 5. Response Time to Interrupts in µC/OS

The distribution of aperiodic interrupt response time for different workload sizes for µC/OS configurations for the benchmark ADPCM decode is shown. Each data point is represented as a circle whose area is proportional to the probability of the response occurring at that point.

NOS: For NOS, the processor utilization has been divided into six categories. These categories include Scheduling, Processing Release Times i.e. inserting a task onto the pause queue or timeout queue, Polling i.e. selecting a task to run while idling, Timer Interrupt Overhead, Processing Clock Ticks and Miscellaneous. As seen in the graphs, the RTOS processor utilization is not quite as large as it is for µC/OS, however, it is still significant enough to be worth optimizing.

Polling is the processing done to call the highest priority task's function, if any or to enter idling otherwise. It occurs after a task completes and when the processor is idle, after every timer interrupt. Unfortunately, the RTM can increase the processing time consumed by the polling operations by up to 25%. This is because without the RTM, polling is as simple as looking at the head of the ready queue. The RTM-based approach has an additional overhead associated with communicating the request to the RTM. The category of processing clock ticks execute at the same instances as polling. But in this case, the RTM reduces the magnitude of the overhead by around 85%.

Both scheduling and processing the release times utilizations increase quadratically with the system load for systems that do not use the RTM. This is because of the linear dependence of their computational complexity and frequency of invocation on the workload. The utilization is constant and trivial for systems with the RTM and it is not even visible on some graphs. The bulk of the lowering of the overhead comes from the nearly 98% reduction in scheduling overhead. The basic RTOS operations that the RTM implements result in the processing overhead required by NOS to be reduced by 20% to 65%.

7.2 Response Time

µC/OS-II: As seen in the graphs, the majority of the response time measurements are a constant value of 1.8 microseconds without the RTM and 1.4 microseconds with the RTM. This is because µC/OS is a preemptive RTOS and it responds to interrupts right away. Deviations from the main value are due to the effects of disabling interrupts while responding to timer interrupts and aperiodic interrupts. When the RTM is not used, the execution time of the timer interrupt ISR is dominated by the time management operation, which scales linearly with the number of tasks. The effect on the response time, as seen in the graphs, is a uniform distribution of values, ranging from the common 1.8 microsecond value to an upper limit, that increases with system load, (up to 11.8 microseconds for the loads studied). However, by using the RTM, the time management operation is always performed in trivial time. The effect of the timer interrupt on response time when using the RTM is a uniform distribution of measurements, ranging from 1.4 microseconds to 2.2 microseconds, an 83% decrease on average.

NOS: Unlike with µC/OS, the response time with NOS has little to do with the effects of the RTOS. Instead, the biggest determining factor has to do with the application code. During the execution of a task, responses to interrupts cannot occur until that task completes. Thus the response time numbers are a uniform distribution of values from the minimum idling response time of 1.4 microseconds to some limit that is only dependent upon the benchmark.

For most applications NOS, being a non-preemptive OS, has virtually no effect on the response time. It is the duration of the processing task and the system load that dictate the response time. However, for small applications whose task invocations finish quickly, NOS does have an effect on response time, and the RTM is able to eliminate this increase.

8. CONCLUSIONS

In this paper we presented the Real-Time Task Manager (RTM)—a hardware module that implements a few common RTOS operations: task scheduling, time management, and event management. By exploiting the inherent parallelism of these operations, the RTM is able to complete them in trivial time, thereby minimizing RTOS overhead. It decreases the processing time used by the RTOS by up to 90% and decreases the maximum response time by up to 81%.

9. ACKNOWLEDGEMENTS

The work of Brinda Ganesh and Paul Kohout was supported in part by NSF grant EIA-9806645 and NSF grant EIA-0000439. The work of Bruce Jacob was supported in part by NSF CAREER Award CCR-9983618, NSF grant EIA-9806645, NSF grant EIA-0000439, DOD award AFOSR-F496200110374, and by Compaq and IBM.

10. REFERENCES

[1] J. Adomat, J. Furunäs, L. Lindh, and J. Stärner. "Real-Time Kernel in Hardware RTU: A step towards deterministic and high performance real-time systems." In *Proceedings of*

Eighth Euromicro Workshop on Real-Time Systems, L'Aquila, Italy, June 1996, pp. 164-168.

[2] M. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[3] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. “The Performance and ENergy Consumption of Three Embedded Real-Time Operating Systems.” In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES ’01)*, Atlanta, GA, November 2001, pp. 203-210.

[4] R. Dick, G. Lakshminarayana, A. Raghunathan, and N. Jha. “Power Analysis of Embedded Operating Systems.” In *Proceedings of the 37th Design Automation Conference*, Los Angeles, CA, June 2000.

[5] International Technology Working Group. *International Technology Roadmap for Semiconductors 2001 Edition: Executive Summary*. Semiconductor Industry Association, 2001.

[6] J. Labrosse. *MicroC/OS-II: The Real-Time Kernel*. R & D Books, Lawrence, KS, 1999.

[7] C. Lee, M. Potkonjak, and W. Mangione-Smith. “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems.” In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO ‘97)*, Research Triangle Park, NC, December 1997.

[8] J. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ, 2000.

[9] J. Mulder, N. Quach, and M. Flynn. “An Area Model for On-Chip Memories and its Application.” *IEEE Journal of SOLID-STATE Circuits*, Vol. 26, No. 2, February 1991, pp. 98-106.

[10] Texas Instruments. *Code Composer Studio User’s Guide*. February 2000.

[11] J. Turley and H. Hakkarainen. “TI’s New ‘C6x DSP Screams at 1,600 MIPS.” *The Microprocessor Report*, Vol. 11, 1997, pp. 14-17.

[12] V. J. Mooney and D. M. Blough, "A Hardware-Software Real-Time Operating System Framework for SOCs," IEEE Design and Test of Computers, pp. 44-51, November-December 2002.

[13] B. E. S. Akgul and V. J. Mooney, "The System-on-a-Chip Lock Cache," International Journal of Design Automation for Embedded Systems, 7(1-2) pp. 139-174, September 2002.

[14] V. Mooney and G. De Micheli, "Hardware/Software Codesign of Run-Time Schedulers for Real-Time Systems,"*Design Automation of Embedded Systems*, 6(1), pp. 89-144, September 2000.

[15] T. Coopee. “Embedded Intelligence.” *InfoWorld*, November 17, 2000.