Real-Time Scheduling with Hardware Data Structures

Gedare Bloom, Gabriel Parmer, Bhagirath Narahari, and Rahul Simha
Department of Computer Science, The George Washington University
Email: {gedare,gparmer,narahari,simha}@gwu.edu

***Abstract*—Two essential features of a real-time operating system (RTOS) are time management and task scheduling. Such features reduce software developers' burden of designing, implementing, and validating generic system infrastructure, thus lowering costs and decreasing application time-to-market. However, there is a cost that is often paid as system overhead during the runtime. Hardware coprocessors that encapsulate RTOS services can reduce system overheads and increase the amount of CPU time available to applications.**

Prior work in scheduling coprocessors have moved scheduling and event processing of a RTOS into hardware. Our work returns the control logic of scheduling coprocessors back to software, and captures the data-centric logic as a *hardware data structure*. Separating the control and data aspects of scheduling coprocessors yields efficient yet flexible hardware support for real-time systems. We demonstrate the flexibility of the hardware data structure by implementing two classic periodic task schedulers, the rate monotonic (RM) and earliest deadline first (EDF) algorithms, and use the same structure for managing timers.

I. Introduction

System overhead must be accounted properly when designing a real-time system, or else task deadlines might be missed. However, accounting for the overhead may lead developers to discard sets of tasks that would otherwise meet their deadlines. Despite the efforts of a real-time operating system (RTOS) to have minimal and constant-time overheads, the demand for high-resolution timing and scheduling by real-time applications, such as video processing, leads to smaller OS ticks and greater scheduling overheads [1].

Prior work has shown that migrating the scheduler to hardware can lower the runtime overheads of admission control [2] and dynamic-priority scheduling [3], [4]. In addition to migrating scheduler services, others have proposed placing timer tick and interrupt processing together with the scheduler in a hardware coprocessor [1], [5], [6]. These hardware task schedulers use some form of a hardware priority queue (PQ) [7] to efficiently select the next schedulable task. Hardware timer management uses delay counters that are updated on each OS tick (now generated by an external interrupt from either the hardware clock or the OS). Our work balances the strengths of hardware with the flexibility of software by retaining the efficient hardware PQ mechanism as a *hardware data structure* (HWDS) and allowing software to control the scheduling policy.

We compare our HWDS-based task scheduling with both software- and hardware-only scheduling, in which the scheduler is fixed in software or hardware respectively. We have implemented each of the three approaches in a cycle-accurate simulator and evaluate them in terms of performance. Our results show that the critical overhead of the scheduler and time tick processing is captured by the HWDS, so that our approach is around 60% as effective as hardware-only scheduling coprocessors without sacrificing the flexibility of software.

The contribution of this work is to introduce HWDSs for improving the performance of real-time systems without sacrificing flexibility. We demonstrate the flexibility of the HWDS by re-using the same hardware for multiple purposes, including runqueue processing for various scheduling polices and timer queue processing.

II. Hardware Data Structures

Hardware support for real-time scheduling is a well-established field, which we review in section V. The key departure that our solution makes with respect to prior work is to isolate the data-centric mechanisms of the hardware from the policy-related hardware control logic. We refer to the data-centric mechanism as a hardware data structure (HWDS), which is related to the notion of data structure. Just as a data structure encapsulates data and its access patterns, so too does a HWDS. By extracting parallelism and performance in the data-centric aspects of task scheduling algorithms, HWDSs improve predictability and performance with respect to software scheduling and improve both flexibility and hardware costs with respect to hardware scheduling.

Our work focuses on preemptive priority-driven scheduling with the rate monotonic (RM) and earliest deadline first (EDF) algorithms for tasks that are independent and periodic (no aperiodic or sporadic tasks) on a single processor, the CPU. Both the RM and EDF algorithms are straightforward to implement in software and have simple schedulability tests. Replacing the ready queue structure of either algorithm with a hardware priority queue (PQ) yields a scheduler that has constant-time operations for adding and removing tasks. *Deadline folding*[8] with modular arithmetic solves the problem of finite deadline values.

A. Hardware Priority Queues

A priority queue is an abstract data structure with insert, extract, and read first (peek) operations.Any data structure that sorts its elements can implement a PQ, for example heaps and self-balancing binary search trees both implement a PQ with logarithmic-time insertion and extraction, with a constant-time cost to read the highest priority element. Hardware PQs are hardware implementations of the PQ data structure.

III. Experimental Setup

To evaluate the effectiveness of HWDS for scheduling, we implemented the RM and EDF scheduling algorithms as software-only schedulers, software schedulers supported by HWDSs, and hardware-only schedulers. We implemented the software schedulers in RTEMS [9] and implemented the hardware support for scheduling in the cycle-accurate Opal processor simulator, a module for the Simics simulator from the GEMS [10] simulation suite. All of our experiments are run with Simics and the GEMS' Opal simulator, without using the Ruby memory model. We also used Cacti 4.1 [11] to estimate the delay and power dissipation of our HWDSs.

A. RTEMS

The Real-Time Executive for Multiprocessing Systems, or RTEMS, is an open source real-time operating system. We extended RTEMS to support our experiments. So that RTEMS will run on GEMS, we added support for the SPARC-V9 family of processors, in particular the UltraSPARC-III processor model for Simics' Serengeti target. We also added an EDF scheduler to RTEMS, and re-wrote the existing scheduler to better isolate the following data structures from the task management logic.

The *timer chain* is a doubly-linked list of zero or more nodes for managing task timers. A task can add a timer with an event to the timer chain and provide a timeout, measured in OS ticks, at which point the timer will "fire" the event. The timer chain is a sorted linked list with $O(n)$ insert, but $O(1)$ removal and efficient updates.

The *ready queue* (or ready chains) is used by the scheduler to manage the set of ready tasks and assign the highest priority task to the CPU. The ready queue is implemented as a 256-element array of FIFO lists. Each list represents a priority level, with the zero level as the highest priority. Tasks with equal priorities are placed on the same FIFO list. The highest priority task is the head of the first non-empty FIFO from the beginning of the ready queue. This structure is efficiently indexed by maintaining a bit map to index non-empty FIFOs.

Periodic tasks are implemented in applications by registering a task-specific timer to track the task's period. A task becomes periodic by creating a timer and executing a loop that starts by setting its timer to the current tick plus its period. We extended the interface for creating periodic tasks so that periodic tasks can be created and scheduled according to the EDF algorithm, which required an extra call-out to update the deadline of a task when a job is released.

Our EDF implementation is straightforward. We replaced the ready queue with a red-black tree that sorts tasks by deadline values. We chose a self-balancing binary search tree over a heap so that duplicate deadlines are detected easily. The ready queue also maintains a linked list which holds all of the ready tasks, including those with duplicate deadlines, which simplifies the search tree implementation.

We use RTEMS with the SPARC-V9 port on the Simics Serengeti target at the 150 MHz CPU frequency, which can provide about 3 MB to an application for stack and heap data. The MMU on the platform cannot be disabled, and RTEMS relies on the firmware to manage the MMU and some of the other system traps. The clock driver relies on the tick register, which is advanced after every instruction by Simics.

B. Task Set Generation

We generate pseudo-random task sets to exercise our scheduler implementations using distributions inspired by Baker [12]. A set of n tasks is created by choosing integer task periods p_i uniformly from $[1, 50]$. Task utilizations u_i are then chosen uniformly from $[0.001, 1)$, implicitly selecting task execution times e_i. After all n tasks have been assigned a utilization, each u_i is normalized so that $\sum_{i=0}^{n} u_i = U$, where U is some target utilization value. This method of generating tasks provides a variety of task sets while being able to control the number of tasks and the task set utilization.

We developed a basic test application for all of our experiments. The test application supports a variable number of independent, periodic tasks with a u_i and p_i. Each task also knows the maximum of all the p_i, $P = MAX_{k=1}^{n}(p_k)$. Each periodic task's workload is a CPU-bound busy loop that counts the number of instructions executed in the loop. The busy loop approximates the number of instructions in a microsecond in an inner loop, and an outer loop counts the number of microseconds of execution to reach u_i in the task's p_i. The busy loop consumes CPU time proportional to u_i, neglecting cache, interrupt, and exception events. Each task t_i executes its periodic loop until it completes $2 * P/p_i$ periods, so that the task with the largest period executes exactly twice, and the test runs for no longer than $3 * (P - 1)$.

We do not include task creation, initialization, or deletion in our measurements. These operations often are not on the critical path and do not make much use of the ready queue or timer chain structures, so we chose to avoid including them in our experiments. We also try to limit the effects of exceptions and interrupts.

C. Hardware Data Structures for Scheduling

We implemented the hardware support for scheduling by modifying Opal and RTEMS. New "magic" instructions trigger the hardware, with different instructions to identify the operations of enqueue, extract, and read (first) for the hardware PQ. We use the software implementation to simulate the functionality of the hardware PQ, and use the magic instructions to properly account for the resources consumed during the hardware execution.

Because Opal does not control the SPARC's tick register, which is used to track time in RTEMS, the hardware operations consume the same amount of perceived time as the software; a welcome side-effect is that the hardware-supported tests execute similar instruction counts and mixes as the software-only tests, making test runs between the software and hardware scheduling consistent.

We model the hardware PQ as a cache, so that we can use freely available tools to obtain reasonable estimates for delay and power costs at a given technology node size. We used Cacti 4.1 [11] to estimate the delay and energy use of enqueue, extract, and read operations on a hardware PQ. For the enqueue and extract operations, we use a fully associative 1KB cache with an 8 byte cache line at the 0.8 μm technology feature size. This is the feature size used by Opal for Wattch power modeling. The 8 byte cache line is sufficient to hold a pointer to a task control block, and the cache tag can hold the priority value. The fully associative cache is a good substitute for the enqueue and extract operations, which cause comparisons at every node in the PQ. The access times and energy for the enqueue and extract operations are 9.2 ns (2 cycles) and 20.29 nJ. Because the PQ can return the highest priority element without using any global wires, the read operation is similar to accessing a direct mapped cache. The same cache parameters are applied but with a direct mapping. The read operation has an access time of 4.74 ns (1 cycle) and uses 0.7 nJ.

All hardware operations were simulated and accounted to a specific HWDS, with separate counters for each operation on the timer chain and ready queue. Each distinct access of a HWDS increments a counter for that structure and for the operation. These counters are used to add the cycle delays and to estimate the dynamic power dissipation of the HWDS. We do not estimate the static power of the hardware PQ.

D. Hardware-only Scheduler

We also simulate a hardware scheduler by encapsulating all of the processing that the system does for the OS clock tick. All explicit accesses of the timer chain, ready queue, and scheduling subsystem, except for yields and dispatches, are subsumed by the hardware scheduler. The hardware scheduler does not capture all task state management, in particular calls made by tasks that modify their state, for example sleeping.

IV. Experiments and Results

We conducted a series of experiments to evaluate HWDSs in the context of real-time systems. The effect of HWDSs on overall system performance is measured by observing the change in CPU cycles when running the basic test described above with varying numbers of tasks and utilizations per task set. We measured performance speedup for software scheduling, HWDS-supported scheduling, and hardware scheduling for both RM and EDF scheduling. In this paper, performance speedup is computed as the difference in cycles consumed between software scheduling and HWDS scheduling, divided by the total number of cycles used to complete the test. Additional measurements are taken to evaluate the energy cost of the HWDS, which is the energy use of HWDS divided by the total processor power dissipation.

A. Performance of Hardware Data Structures

We measured the performance speedup from using HWDSs by generating task sets of size n in 20, 40, 60, and 80. We normalized the utilizations of tasks to a task set utilization U of 0.2, 0.4, 0.6, and 0.8. 10 task sets of each combination of n and U were generated, for a total of 160 task sets in all. Each task set is a copy of the basic test, with pseudo-random periods and utilizations. We ran the same task sets with the software scheduler, the software scheduler augmented with HWDSs, and the hardware scheduler.

Figure 1 shows the performance speedup of using hardware PQs for replacing the software-based timer chain and ready queue for the RM and EDF schedulers. The results are grouped according to the target task utilization U, with each bar an average of the 10 tasks for the particular n and U. The first four bars are for $U = 0.4$, the next four are $U = 0.6$, then $U = 0.8$, and finally $U = 1.0$. Error bars show the standard deviation.

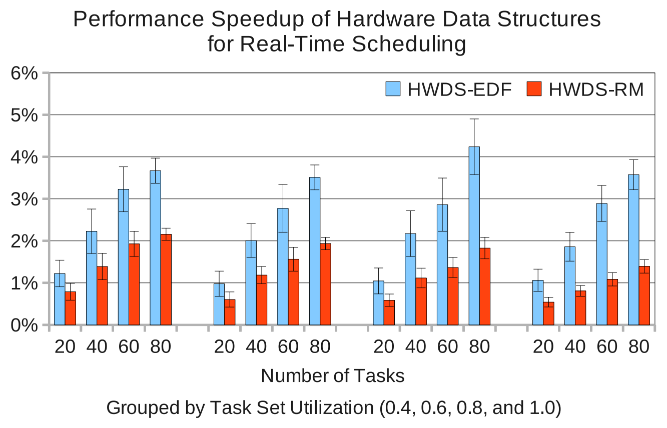

Fig. 1. Performance improvement of the hardware data structure (HWDS) based RM and EDF schedulers over software-only scheduling. Improvements come from replacing both the timer chain and ready queue with a hardware priority queue.

Part of the motivation for HWDSs is that they provide similar performance benefits to a hardware-based scheduler while being generic, flexible, and applicable to many different algorithms. Figure 2 shows initial results for how well the HWDSs perform with respect to our approximation of a hardware scheduler. We only add the delay of accessing the HWDS to the hardware scheduler and assume other operations are not on any critical paths. Also, we ignore possible communication latencies, which can be significant for off-chip coprocessor-based hardware schedulers. Thus our comparison is at least fair, and at worst biased in favor of the hardware scheduler.

Across all of the tests, the gap between the HWDS approach and the hardware scheduler sits between 0.3% and 3.8% of overall processor performance speedup. This is a modest gap when one considers the conservative estimates we use for the performance costs of the hardware scheduler.

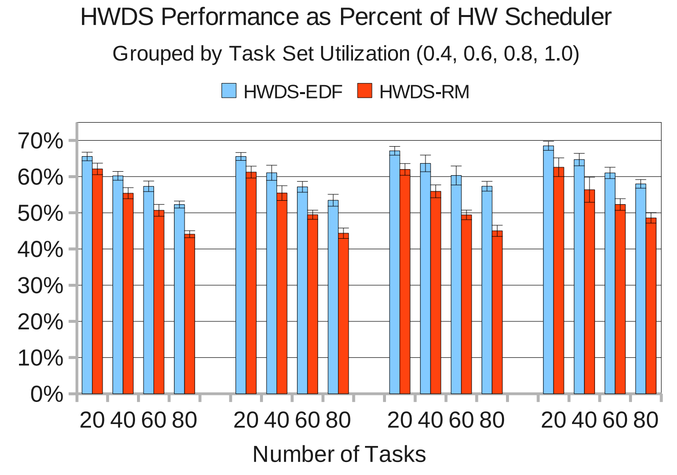

Fig. 2. Performance speedup of the HWDS as compared to the speedup of the hardware-only RM and EDF schedulers over software-only scheduling. Performance penalties for accessing the HWDSs and the hardware schedulers are assumed to be identical, which is unlikely the case.

V. Related Work

Hardware support for scheduling has been an area of interest in the queuing hardware of packet-switched networks. Moon et al. [7] compare four approaches to hardware PQs for high-speed networks and introduce an approach that melds two of the previous solutions. Kim and Shin [8] describe an architecture for EDF scheduling for ATM switch networks, which is also applicable to task scheduling and introduces deadline folding.

For real-time task scheduling systems a number of hardware scheduling coprocessors have been proposed. The first use of hardware scheduling of which we are aware is in the Spring Scheduling Coprocessor (SSCoP) [2]. SSCoP is primarily used to generate a schedule for a set of tasks and to ensure that the schedule is feasible under the system's real-time constraints. Saez et al. [3] put EDF and slack stealing with task state management in hardware. Hildebrandt et al. [4] propose enhanced least-laxity-first scheduling. Kuacharoen et al. [1] implement a configurable hardware scheduler that manages sleeping tasks and the task table. Kohout et al. [5] propose the Real-Time Task Manager (RTM), which is a processor extension that implements task management in hardware, including fixed priority scheduling, timer management, and event management. Zong [6] implements EDF scheduling and task state management in hardware for μC/OS.

In contrast to the related work, our approach implements only the PQ mechanism in hardware and allows RTOS software to control the scheduling policy. Our approach is flexible and generic like software while remaining fast like hardware.

VI. Conclusion and Future Work

As a hybrid approach, hardware data structures provide the flexibility of software and the performance of hardware. Initial results are encouraging, showing that HWDS capture at least 50% of the performance benefits of scheduling coprocessors.

We are pursuing multiple directions with this work. First, we focused on overall performance so far, but we are also interested in how HWDS affects the latency of OS services and thus schedulability. Second, our model of the HWDS and hardware schedulers suffers from software artifacts such as exceptions and cache misses, which we plan to reduce. Third, a HWDS affects the system's memory usage. We are investigating how a HWDS changes the cache behavior of memory-bound applications. Fourth, our method for generating test applications is based on prior work in schedulability and may not be appropriate for our uses; we are interested in generating test applications that can be used to reliably evaluate HWDS in the context of real-time scheduling.

Acknowledgment

This work is supported in part by NSF grant CNS-0934725 and AFOSR grant FA9550-09-1-0194. The authors thank Guru Venkataramani and Veronica Bloom for their insightful suggestions.

References

[1] P. Kuacharoen, M. A. Shalan, and V. J. M. III, "A configurable hardware scheduler for Real-Time systems," *In Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp. 96—101, 2003. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.58.585
[2] W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J. A. Stankovic, G. Wallace, and C. Weems, "The spring scheduling coprocessor: a scheduling accelerator," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 7, no. 1, pp. 38–47, 1999. [Online]. Available: http://portal.acm.org/citation.cfm?id=297731.297736
[3] S. Saez, J. Vila, A. Crespo, and A. Garcia, "A hardware scheduler for complex real-time systems," in *Industrial Electronics, 1999. ISIE '99. Proceedings of the IEEE International Symposium on*, vol. 1, 1999, pp. 43–48 vol.1.
[4] J. Hildebrandt, F. Golatowski, and D. Timmermann, "Scheduling coprocessor for enhanced Least-Laxity-First scheduling in hard Real-Time systems," in *Real-Time Systems, Euromicro Conference on*. Los Alamitos, CA, USA: IEEE Computer Society, 1999, p. 0208.
[5] P. Kohout, B. Ganesh, and B. Jacob, "Hardware support for real-time operating systems," in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. Newport Beach, CA, USA: ACM, 2003, pp. 45–51. [Online]. Available: http://portal.acm.org/citation.cfm?id=944645.944656
[6] L. Zong, "Nanoprocessors: Configurable hardware accelerators for embedded systems," Master's Thesis, 2003.
[7] S. Moon, K. Shin, and J. Rexford, "Scalable hardware priority queue architectures for high-speed packet switches," in *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, 1997, pp. 203–212.
[8] B. K. Kim and K. Shin, "Scalable hardware earliest-deadline-first scheduler for ATM switching networks," in *Real-Time Systems Symposium, IEEE International*. Los Alamitos, CA, USA: IEEE Computer Society, 1997, p. 210.
[9] "RTEMS: Real-Time executive for multiprocessor systems." http://www.rtems.com/. [Online]. Available: http://www.rtems.com/
[10] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1105734.1105747
[11] D. Tarjan, S. Thoziyoor, and N. P. Jouppi, "CACTI 4.0," HP Laboratores Palo Alto, Tech. Report HPL-2006-86, 2006.
[12] T. P. Baker, "A comparison of global and partitioned EDF schedulability tests for multiprocessors," Florida State University, Tech. Rep. TR-051101, 2005.