Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2012) 1–9
Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Improving System Predictability and Performance via Hardware Accelerated Data Structures

Chetan Kumar N G[a], Sudhanshu Vyas[a], Jonathan A. Shidal[b], Ron K. Cytron[b], Christopher D. Gill[b], Joseph Zambreno[a], Phillip H. Jones[a,1]

[a]*Iowa State University, Department of Electrical and Computer Engineering, Ames, IA*
[b]*Washington University, Computer Science and Engineering, St. Louis, MO*

Abstract

In Dynamic Data-Driven Application Systems, applications must dynamically adapt their behavior in response to objectives and conditions that change while deployed. One approach to achieve dynamic adaptation is to offer middleware that facilitates component migration between modalities in response to such dynamic changes. The triggering, planning, and cost evaluation of adaptation takes place within a scheduler. Scheduling overhead is a major limiting factor for implementing dynamic scheduling algorithms with high frequency timer-tick resolution in real time systems. In this paper, we present a scalable hardware scheduler architecture for real time systems that reduces processing overhead and improves timing predictability of the scheduler. A new hardware priority queue design is presented, which supports insertions in constant time, and removals in $O(\log n)$ time. The hardware scheduler supports three (Rate Monotonic Scheduling (RMS), Earliest Deadline First (EDF), and priority based) scheduling algorithms, which can be configured during run-time. The interface to the scheduler is provided through a set of custom instructions as an extension to the processors instruction set architecture. We also report on our experience migrating between two implementations of an ordered-set implementation, with the goal of providing predictable performance for real-time applications.

Keywords: hardware scheduler, real-time system, priority queue, ordered set, hardware accelerated data structure

1. Introduction

In the context of Dynamic Data-Driven Applications Systems (DDDAS), we have been investigating data structure implementations that are suitable for avionics missions with multimodal dynamic requirements. These data structures serve DDDAS through their ability to adapt to evolving conditions and change their behavior to preserve an application's current mission or to facilitate migration to a new mission. In particular, we are interested in applications where elements of surprise may impose sudden and perhaps short-lived modality shifts. For example, a component of an application that has been operating under best-effort conditions may be required to respond in real-time based

Email addresses: ckng@iastate.edu (Chetan Kumar N G), spvyas@iastate.edu (Sudhanshu Vyas), shidalj@wustl.edu (Jonathan A. Shidal), cytron@cse.wustl.edu (Ron K. Cytron), cdgill@cse.wustl.edu (Christopher D. Gill), zambreno@iastate.edu (Joseph Zambreno), phjones@iastate.edu (Phillip H. Jones)
[1]Corresponding author

on emergence of a threat or environmental degradations. The modalities we currently consider are best-effort (high performance), real-time, and embedded (small footprint). The general idea is that an implementation while operating in one mode can respond to a request to change its mode. The response includes not only the data structure's initial movement toward the new mode, but also a schedule indicating its projected performance as it switches modality.

The data structures we develop in this manner are "elastic" in the sense that their functionality does not change, but their implementations adapt between the modes under consideration. An earlier and very specific example of our work is a hashtable implementation that is suitable for real-time [1].

While we are investigating algorithmic solutions to achieve elastic data structures, we focus in this paper on another technique for obtaining real-time implementations, namely the development of logic deployed in hardware to achieve predictability and improve performance. The subject of our study here is a *priority queue* and its use within a real-time operating system to facilitate scheduling.

A real-time operating system (RTOS) is designed to execute tasks within given timing constraints. An important characteristic of an RTOS is predictable response under all conditions. The core of the RTOS is the scheduler, which ensures tasks are completed by their deadline. The choice of a scheduling algorithm is crucial for a real-time application. Online scheduling algorithms incur overhead, as the task queues must be updated regularly. This action is typically paced using a timer that generates periodic interrupts. The scheduler overhead generally increases with the number of tasks. A high resolution timer is required to distribute CPU load accurately based on a scheduling discipline in real-time systems, but such fine-grain time management increases the operating system overhead [2], [3].

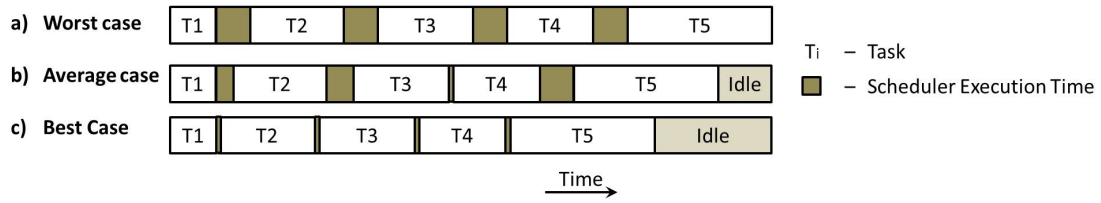

Figure 1: In order to allow analytical analysis of schedule feasibility, worst-case execution time (WCET) typically needs to be assumed. Thus, scheduler execution time variations that cause large differences between WCET and typical case execution time reduce utilization of system computing resources.

The extent to which a scheduler can ideally implement a given scheduling paradigm (e.g. EDF, RMS), and thus provide the guarantees associated with that paradigm, is in part dependent on its timing determinism. A metric for helping quantify the amount of non-determinism that is introduced to the system by the scheduler is the variation in execution time among individual scheduler invocations. This can be roughly summarized by noting its best-case and worst-case execution times. Variations in scheduler execution time can be caused by system factors such as changes in task set composition, cache misses, etc. Hence, reducing the scheduler's timing sensitivity to such factors can help increase deterministic behavior, which in turn allows the scheduler to better model a given scheduling paradigm.

Figure 1 illustrates how the variation in scheduler overhead affects processor utilization. To ensure that tasks meet their deadlines, the scheduler's worst-case execution times are often overestimated. This can cause a system to be underutilized and wastes CPU resources. In this paper, we examine how the scheduler overhead and its variation can be reduced by migrating scheduling functionality (along with time-tick processing) to hardware logic. The expected results of our efforts are increased CPU utilization and better system predictability. Another benefit is that the hardware clock provides accurate high-resolution timing.

The rest of the paper is organized as follows. Section 2 presents related work on hardware schedulers. Section 3 describes the scalable hardware scheduler architecture and implementation details. The evaluation methodology and results are discussed in Sections 4 and 5. Section 6 describes a software approach to an adaptive ordered-set data structure. Conclusions and future work are presented in Section 7.

2. Related Work

Many architectures [3], [4], [5], [6], [7], [8] have been proposed to improve the performance of schedulers using hardware accelerators. A real time kernel called FASTHARD has been implemented in hardware [3]. The scheduler of FASTHARD can handle 256 tasks and 8 priority levels. The Spring scheduling coprocessor [4] was built to accelerate scheduling algorithms used in the Spring kernel [9], which was used to perform feasibility analysis of the schedule. Mooney et al. [5] implemented a configurable hardware scheduler that provided support for three scheduling disciplines, configurable during runtime. A slack stealing scheduling algorithm was implemented in hardware [6] to support scheduling of tasks (periodic and aperiodic) and to reduce scheduling overhead. A hardware scheduler for multiprocessor system on chip is presented in [7], which implements the Pfair scheduling algorithm. A real time task manager (RTM) that implements scheduling, time management, and event management in hardware is presented in [8]. That RTM supports static priority-based scheduling and is implemented as an on-chip peripheral that communicates with the processor though memory mapped interface.

Most of the schedulers listed above implement some kind of priority based scheduling algorithm that requires a priority queue to sort the tasks based on their priority. Many hardware priority queue architectures have been implemented in the past, most of them in the realm of real-time networks for packet scheduling [10, 11, 12]. Moon et al. [10] compared four scalable priority queue architectures: fifo, binary tree, shift registers and systolic array based. The shift-register architecture suffers from bus loading, as new tasks must be broadcasted to all the queue cells. The systolic array architecture overcomes the problem of bus loading at the cost of doubling hardware storage requirements. The hardware complexity for both the shift register and systolic array architecture increases linearly with the number of elements, as each cell requires a separate comparator. This makes these architecture expensive to scale in terms of hardware resources. Bhagwan and Lin [11] proposed a new pipelined priority queue architecture based on p-heap (a new data structure similar to binary heap). A pipelined heap manager was proposed in [12] to pipeline conventional heap data structure operations. Both of these pipelined implementations of a priority queue are scalable and are designed to achieve high throughput, but at the expense of increased hardware complexity.

In this paper we present a scalable hardware priority queue architecture that implements a conventional binary heap in hardware. The priority queue is used as a part of the scheduler to improve system performance and predictability. The hardware priority queue supports constant time enqueue operations and dequeue operations in $O(\log n)$ time. The hardware utilization of the our priority queue increases logarithmically with the queue size and avoids complex pipelining logic.

3. Architecture Overview

The hardware scheduler architecture we propose is designed to reduce time-tick processing and scheduling overhead of the system. The design also uses concurrency in hardware to make the operations on a priority queue more predictable. The instruction set architecture of the processor is correspondingly extended to support a set of custom instructions to communicate with the scheduler. The hardware scheduler executes the scheduling algorithm and returns the control to the processor along with the next task to execute, and context switching is then done in software. A software timer periodically generates interrupts to check for the availability of a higher priority task. The check is accomplished using a single custom instruction that returns a preempt flag set by the hardware scheduler, based on which the processor can then choose to continue the execution of the current task or to run another. A high level block diagram of the hardware scheduler is shown in Figure 2.

The controller is the central processing unit of the scheduler. It is responsible for the execution of the scheduling algorithm. The controller processes instruction calls from the processor and monitors task queues. The timer unit keeps track of time elapsed since the start of the scheduler. This provides accurate high-resolution timing for the scheduler. The resolution of the timer-tick can be configured at runtime. The interface to the scheduler is provided through a set of custom instructions as an extension to the instruction set architecture of the processor. This removes bus dependencies for data transfer. Basic scheduler operations such as run, configure, add task, and preempt task are supported. The ready queue stores active tasks based on their priority. The sleep queue stores sleeping tasks until their activation time. The task with the earliest activation time is at the front of the sleep queue. At the core of the scheduler are the task queues, which are implemented as priority queues that keep the tasks in sorted order based on their priority (ready queue) or activation time (sleep queue).

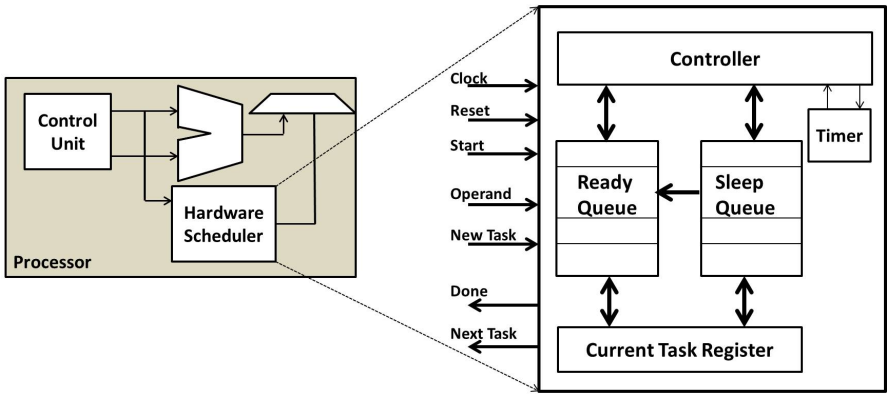

Figure 2: A high level architecture diagram of the hardware scheduler along with the custom instruction interface.

3.1. Priority Queue Architecture

One of the common software data structures for implementing a priority queue is the binary heap, which supports enqueue and dequeue operations in $O(\log n)$ time. The binary heap is stored as a linear array where the first element corresponds to the root. Given an index i of an element, $i/2$, $2i$ and $2i + 1$ are the indices of its parent, left and right child respectively. Here we present a hardware implementation of the conventional binary heap that supports enqueue and peek operations in $O(1)$ time and dequeue operations in $O(\log n)$ time. Although the dequeue operation takes $O(\log n)$ time to complete, the top-priority task can be returned immediately, so that a dequeue operation overlaps its work with that of the rest of the scheduler and the application. A high level architecture diagram for the priority queue is shown in Figure 3.

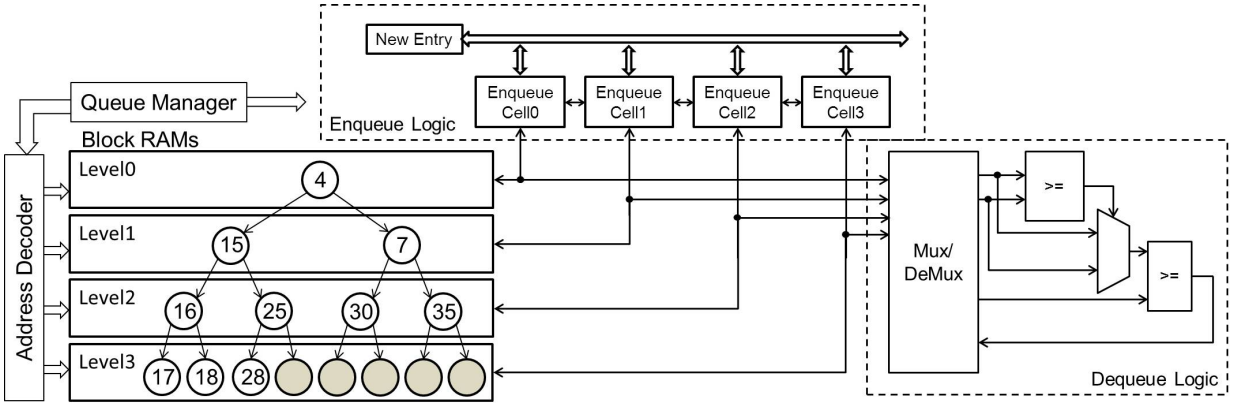

Figure 3: The priority queue architecture.

Central to the priority queue is the queue manager, which provides the necessary interface and executes operations on the queue. Elements in each level of the heap are stored in separate on-chip memories called Block Rams (BRAMs) to enable parallel access to heap elements, similar to [11, 12]. The address decoder generates addresses and control signals for the BRAM blocks. Queue operations are explained in detail below.

3.1.1. Enqueue

Enqueue operations in a binary heap are accomplished by inserting the new element at the bottom of the heap and performing compare-swap operation with successive parents until the priority of the new element is less than its parent. The worst-case behavior occurs when the priority of the new element is greater than the rest of the nodes present in the heap. In this case, the new element bubbles-up all the way to the root of the heap. We first calculate

the path from a leaf node to the root. The leaf node is always one more than the current size of the queue. This path includes all ancestors from the leaf node to the heap's root. The heap property ensures that the elements in this path are in sorted order.

The shift register mechanism, shown in Figure 3, inserts a new element in constant time. This is similar to the shift-register priority queue described in [10]. Each level of the heap is mapped to an enqueue cell, which consists of a comparator, multiplexor and a register. The element to be inserted is broadcast to all the cells during an enqueue operation. The enqueue operation is then completed in the three steps shown in Figure 4. In the first step, all the elements in the path from the leaf node to root node are loaded into the corresponding enqueue cells. The address for each BRAM block is generated by the address decoder. In the second step, the comparator in each enqueue cell compares the priority of the new element with the element stored locally and decides whether to latch the current element, new element or the element above it. In the final step, the elements along with the new entry are stored back into the heap.

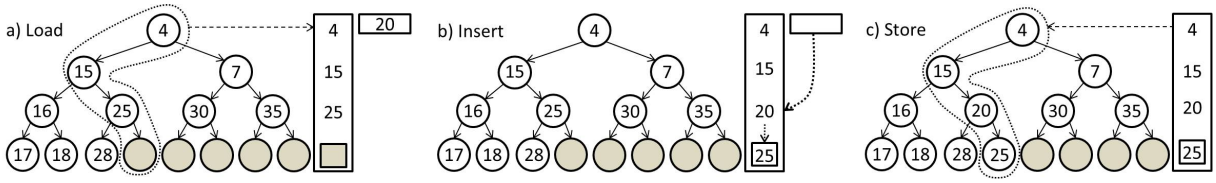

Figure 4: Steps of enqueue operation.

3.1.2. Dequeue

The dequeue operation can be divided into two parts: removing the root element from the queue (as the value to be returned by the dequeue call), and reconstruction of the heap. The root element is removed by replacing it with the last element of the queue to keep the heap balanced. The new root element is then compared with smallest of its children and swapped if the priority of new node is less than that of a child. This operation is repeated until the priority of the new root element is more than that of its children. An example of a dequeue operation is shown in Figure 5

Note that the highest priority value is obtained in constant time and as the priority queue is managed in hardware the processor is not required to wait for the operation to complete. The worst case execution time of a dequeue operation is $O(\log n)$, which would affect the rate at which consecutive operations can be performed on the queue. However, since requests for dequeue operations are paced by software, consecutive dequeue operations on the task queue are rare. Hence, this has little effect on the performance of the scheduler.

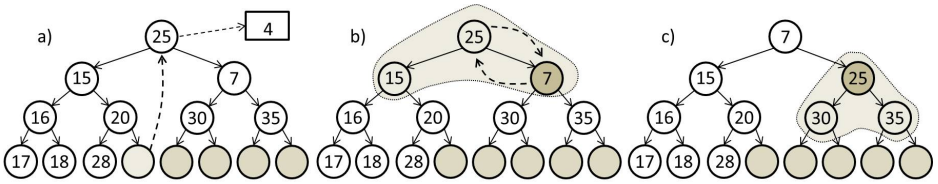

Figure 5: Steps of dequeue operation.

Comparing our approach with the related work reported in Section 2, our approach scales nicely without requiring hardware to manage pipelining and obtains suitably low latencies for the scheduler.

4. Evaluation Methodology

The hardware scheduler was deployed and evaluated on the Recongurable Autonomous Vehicle Infrastructure (RAVI) board, an FPGA development platform developed at Iowa State University. RAVI leverages Field Programmable Gate Array (FPGA) technology to allow custom hardware to be tightly integrated to a soft-core processor

on a single computing device. It enables exploration of the software/hardware codesign space for designing system architectures that best fit an applications requirements. The portions of the RAVI board we used for our experiments included the Cyclone III FPGA, the on-board DDR DRAM and the UART port. The FPGA was used to implement the NIOS-II (Alteras soft-processor), the DDR stored software that was run on the NIOS-II, and the UART port supported data collection. A pictorial description of the setup is shown in Figure 6.

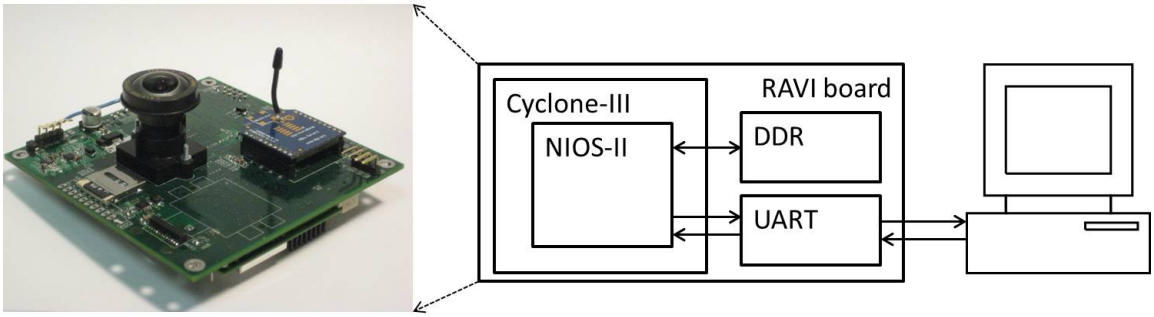

Figure 6: Evaluation platform.

The hardware scheduler is implemented as an extension to the instruction set architecture (custom instruction) of a Nios II embedded processor running at 50 MHz on an Altera Cyclone III FPGA. The scheduler supports up to 256 tasks and can be configured to use EDF or fixed priority based scheduling algorithm such as RMS. A software test bench was built to measure the overhead of the scheduler for different task sets and timer resolutions. An Earliest Deadline First (EDF) scheduler was deployed to measure the impact of running a dynamic scheduling algorithm on the processor. EDF is a dynamic priority-based scheduling algorithm in which higher priorities are assigned to the tasks with closer absolute deadlines. A software EDF scheduler implementation was used to characterize the runtime overhead involved in implementing a dynamic scheduling algorithm and to compare against our hardware implementation.

5. Results and Analysis

The overhead of the scheduler was measured for different sets of tasks and timer tick resolutions. Figure 7 shows the percentage overhead of the software scheduler. As evident in Figure 7, the scheduler overhead increases with the number of tasks and the timer-tick resolution. For a timer tick resolution of 0.1ms and with 256 tasks, the processor overhead reaches up to 18%. This would limit the amount of time available for the CPU and may cause tasks to miss deadlines. Most of this overhead results from time tick processing where the scheduler periodically processes interrupt requests to check for new tasks and managing the task queues. This has been a limiting factor for implementing dynamic priority based scheduling algorithms in embedded real time systems.

Figure 8 shows the scheduling overhead when the hardware scheduler is used. The results show that when the timer tick resolution is set to 0.1ms and with 256 tasks the scheduler overhead is less than 0.5%. This shows a 97% reduction in scheduler overhead as compared to the software model. Most of the scheduling overhead is eliminated by the hardware scheduler, as the time tick processing and a majority of the scheduling functionality is migrated to hardware. A call to the software scheduler is replaced by a custom instruction call to obtain the next task for execution or to preempt the current task. The predictability of the scheduler can be measured as the variation in the execution time of a single call to the scheduler. The best, average and worst case execution times of the scheduler are shown in Figure 9. The difference between the best case and worst case execution time is large in the software scheduler. Hence the scheduler can be a significant source of unpredictability in real time systems. The system then must be designed for the worst case behavior to ensure task deadlines are not missed, which would cause the CPU to be underutilized most of the time. On the other hand, the execution times of the hardware scheduler show more deterministic behavior with very little variation, which results in tighter worst-case execution time bounds.

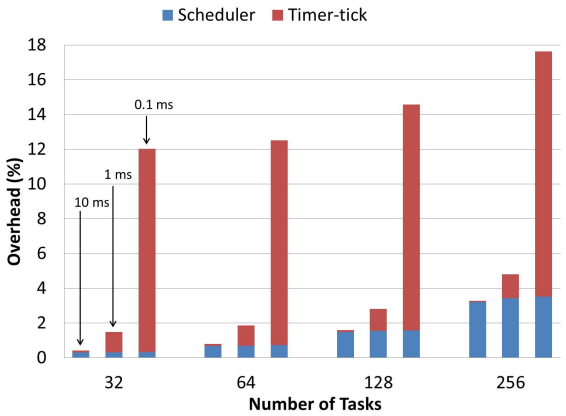

Figure 7: Software scheduler overhead.

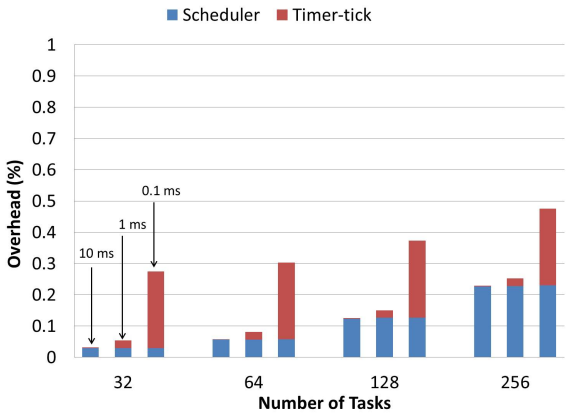

Figure 8: Hardware scheduler overhead.

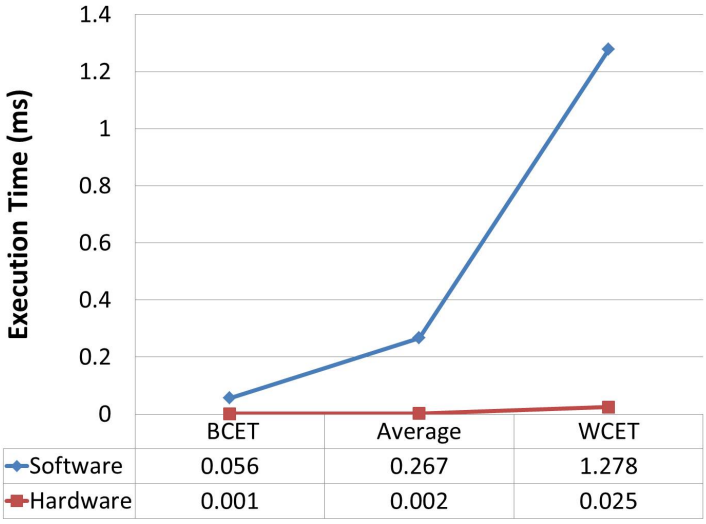

Figure 9: Variation in scheduler execution time.

6. Adaptable Binary Search Trees

Although this paper is primarily concerned with deployment of data structure functionality in hardware to achieve real-time properties, we report here on our recent efforts to achieve predictability by using multiple implementations of an abstract data type in software alone. The problem we address is that of maintaining an *ordered set* using a *binary search tree* (BST). A BST organizes the elements of a set as follows. At each node n, all elements ordered less than n's value are stored in n's left subtree, and all elements ordered greater than n's value are stored in n's right subtree. Thus, an in-order traversal of the tree produces a listing of the set's elements in ascending order.

If a BST is balanced, then all single-node operations are bounded by $O(\log n)$ time, which is the height of a balanced tree of n nodes. The shape of a BST depends on the order in which elements are inserted and deleted from the ordered set. Without care, a BST can become *un*balanced: the most unbalanced tree behaves as a linked list, with all single-node operations taking $O(n)$ time. For example, such a tree results from inserting n elements in ascending order.

Self-balancing BSTs are therefore an important data structure for real-time systems, and we consider here two such implementations: AVL trees and Red-Black trees. The following table summarizes the worst-case behaviors of

interest for these implementations:

Worst case	AVL	Red-Black
Rotations for insert	2	2
Rotations for delete	$\log n$	3
Height for n nodes	$< 1.44 \log (n + 2) - 1$	$\leq 2 \log (n) + 1$

Because of the height bounds, both implementations achieve an $O \log (n)$ time bound to find a node in the tree. While the bounds are asymptotically the same, Red-Black trees are essentially as unbalanced as possible while maintaining their bounds, while AVL trees are as balanced as possible. The difference in subtree height at any Red-Black tree node n can be off by a factor of 2, while the subtree heights for any AVL tree node n differ by at most 1.

While asymptotically irrelevant, these differences can be considerable for real-time applications, and neither implementation is preferable in all situations to the other. As shown in the table above, AVL trees maintain their better lookup performance by performing at most $\log (n)$ rotations in response to changes in the BST. On the other hand, changes to a Red-Black tree preciptate at most 3 rotations, but lookup times could differ by a factor of 2.

We seek an implementation that can dynamically change its behavior between the two implementations in response to DDDAS considerations. Our work thus far has concerned the cost of converting one implementation to the other. We report here on a new technique for converting an AVL tree to a Red-Black tree. One approach is simply to traverse the tree and establish the color at each node. This has been considered by Glick [13], and while that algorithm requires no rotations to establish the Red-Black tree, a traversal of the entire tree is required.

For real-time systems, an operation that must traverse the entire tree is signicantly more expensive than all of the other operations on a BST. To avoid such expense, we observe the following property of establishing a Red-Black tree from an extant AVL tree. The color at a given node can be determined by the height of a node and the height of its parent in the BST. For AVL trees that include such height information, establishing Red-Black coloring can be accomplished incrementally as operations are performed on the BST. Information the algorithm uses to color a node is its height and the height of its parent. To color a node n, three cases must be considered. The first is if the parent of n has even height. In this case we simply color n black. The next two cases occur when the parent of n has odd height. If n has even height it must be colored red, if odd it is colored black. It follows from this construction that it is impossible for both a node and its parent to be colored red.

In a DDDAS system, we can observe the types of functions that are called on the BST and thus predict sections of the tree that may be more active than others. The tree can be converted from AVL to Red-Black according to these predictions. For instance, if the system anticipates a search-heavy section of operations, the system will convert the tree to AVL for faster searches. When the system anticipates new elements will be added and deleted from the tree, it can convert the tree to Red-Black.

Our approach to coloring nodes incrementally creates an intriguing idea of a *hybrid* tree that contains some AVL sections as well as Red-Black sections. This could allow sections of the tree that are being searched often to remain AVL for quicker searches, while other sections are Red-Black.

7. Conclusion

A scalable hardware scheduler has been implemented that supports 256 tasks and can be configured to run one of three (EDF, RMS, other fixed-priority) scheduling disciplines. A new hardware priority queue architecture is implemented that supports enqueue and peek operations in $O(1)$ time, returns the top-priority task in $O(1)$ time, and completes a dequeue operation in $O(\log n)$ time. The hardware scheduler reduced the scheduling and time tick processing overhead of the system. Our results show that the hardware scheduler has reasonably deterministic behavior with predictable execution times, which is necessary in high-performance real time systems.

Acknowledgments

This work is supported in part by the National Science Foundation (NSF) under award CNS-1060337, and by the Air Force Office of Scientific Research (AFOSR) under award FA9550-11-1-0343.

References

[1] S. Friedman, N. Leidenfrost, B. C. Brodie, R. K. Cytron, Hashtables for embedded and real-time systems, in: Proceedings of the IEEE Workshop on Real-time Embedded Systems, 2001.

[2] T. R. Park, J. H. Park, W. H. Kwon, Reducing os overhead for real-time industrial controllers with adjustable timer resolution, in: Industrial Electronics. ISIE. IEEE International Symposium on, 2001, pp. 369–374 vol.1.

[3] J. Adomat, J. Furunas, L. Lindh, J. Starner, Real-time kernel in hardware rtu: a step towards deterministic and high-performance real-time systems, in: Real-Time Systems, 1996., Proceedings of the Eighth Euromicro Workshop on, 1996, pp. 164–168.

[4] W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J. Stankovic, G. Wallace, C. Weems, The spring scheduling coprocessor: a scheduling accelerator, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on (1999) 38–47.

[5] P. Kuacharoen, M. A. Shalan, V. J. M. III, A configurable hardware scheduler for real-time systems, in: in Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, CSREA Press, 2003, pp. 96–101.

[6] S. Saez, J. Vila, A. Crespo, A. Garcia, A hardware scheduler for complex real-time systems, in: Industrial Electronics, 1999. ISIE '99. Proceedings of the IEEE International Symposium on, 1999, pp. 43–48 vol.1.

[7] N. Gupta, S. Mandal, J. Malave, A. Mandal, R. Mahapatra, A hardware scheduler for real time multiprocessor system on chip, in: VLSI Design, 2010. VLSID '10. 23rd International Conference on, 2010, pp. 264–269.

[8] P. Kohout, B. Ganesh, B. Jacob, Hardware support for real-time operating systems, in: Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on, 2003, pp. 45–51.

[9] J. Stankovic, K. Ramamritham, The spring kernel: a new paradigm for real-time systems, Software, IEEE (1991) 62–72.

[10] S.-W. Moon, K. Shin, 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.

[11] R. Bhagwan, B. Lin, Fast and scalable priority queue architecture for high-speed network switches, in: INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, 2000, pp. 538–547 vol.2.

[12] A. Ioannou, M. Katevenis, Pipelined heap (priority queue) management for advanced scheduling in high-speed networks, Networking, IEEE/ACM Transactions on (2007) 450–461.

[13] J. Glick, How to make a red-black tree from an avl tree, `http://cseweb.ucsd.edu/classes/su05/cse100/cse100wa2.txt`.