Hardware-software architecture for priority queue management in real-time and embedded systems

N.G. Chetan Kumar and Sudhanshu Vyas

Department of Electrical and Computer Engineering,
Iowa State University,
2215 Coover Hall, Ames, IA 50011, USA
E-mail: ckng@iastate.edu
E-mail: spvyas@iastate.edu

Ron K. Cytron and Christopher D. Gill

Department of Computer Science and Engineering,
Washington University in St. Louis,
1 Brookings Dr., St. Louis, MO 63130, USA
E-mail: cytron@cse.wustl.edu
E-mail: cdgill@cse.wustl.edu

Joseph Zambreno and Phillip H. Jones*

Department of Electrical and Computer Engineering,
Iowa State University,
2215 Coover Hall, Ames, IA 50011, USA
E-mail: zambreno@iastate.edu
E-mail: phjones@iastate.edu
*Corresponding author

Abstract: The use of hardware-based data structures for accelerating real-time and embedded system applications is limited by the scarceness of hardware resources. Being limited by the silicon area available, hardware data structures cannot scale in size as easily as their software counterparts. We assert a hardware-software co-design approach is required to elegantly overcome these limitations. In this paper, we present a hybrid priority queue architecture that includes a hardware accelerated binary heap that can also be managed in software when the queue size exceeds hardware limits. A memory mapped interface provides software with access to priority-queue structured on-chip memory, which enables quick and low overhead transitions between hardware and software management. As an application of this hybrid architecture, we present a scalable task scheduler for real-time systems that reduces scheduler processing overhead and improves timing determinism of the scheduler.

Keywords: priority queue; hardware-software co-design; real-time and embedded systems; hardware scheduler.

Reference to this paper should be made as follows: Kumar, N.G.C., Vyas, S., Cytron, R.K., Gill, C.D., Zambreno, J. and Jones, P.H. (2014) 'Hardware-software architecture for priority queue management in real-time and embedded systems', *Int. J. Embedded Systems*, Vol. 6, No. 4, pp.319–334.

Biographical notes: N.G. Chetan Kumar is a PhD student in the Department of Electrical and Computer Engineering, where he is working with Prof. Phillip Jones. He completed his BS in Electronics and Communication at Visveswaraya Technological University, Bangalore, India in 2007. His research interests include embedded and real-time systems and hardware/software co-design. His current research focuses on developing techniques to improve predictability in execution of core system operations in real-time systems, using hardware-software co-design approaches.

Sudhanshu Vyas is a graduate student pursuing his PhD at Iowa State University. He joined ISU in the Fall of 2009. Before joining, he received his BE from Birla Institute of Technology in Electronics and Communication Engineering in 2006 and worked at CG-CoreEl, an embedded systems company based in Bangalore. His research interests include reconfigurable architectures, embedded systems, control systems and FPGA fault tolerance.

Ron K. Cytron is a Professor of Computer Science and Engineering at Washington University. His research interests include optimised middleware for embedded and real-time systems, fast searching of unstructured data, hardware/runtime support for object-oriented languages, and computational political science. He has over 100 publications and ten patents. He has received the SIGPLAN Distinguished Service Award and is a co-recipient of SIGPLAN Programming Languages Achievement Award. He served as Editor-in-Chief of *ACM Transactions on Programming Languages and Systems* for six years. He participated in writing the Computer Science GRE Subject Test for eight years and chaired the effort for three years. He is a Fellow of the ACM.

Christopher D. Gill is a Professor of Computer Science and Engineering at Washington University in St. Louis. His research includes formal modelling, verification, implementation, and empirical evaluation of policies and mechanisms for enforcing timing, concurrency, footprint, fault-tolerance, and security properties in distributed, mobile, embedded, real-time, and cyber-physical systems. He developed the Kokyu real-time scheduling and dispatching framework used in several AFRL and DARPA projects and flight demonstrations, and led development of the nORB small-footprint real-time object request broker at Washington University. He has over 60 refereed technical publications and has an extensive record of service in review panels, standards bodies, workshops, and conferences for distributed, real-time, embedded, and cyber-physical systems.

Joseph Zambreno is an Associate Professor in the Department of Electrical and Computer Engineering at Iowa State University, Ames, where he has been since 2006. His research interests include computer architecture, compilers, embedded systems, and hardware/software co-design, with a focus on run-time reconfigurable architectures and compiler techniques for software protection. He was a recipient of a National Science Foundation Graduate Research Fellowship, a Northwestern University Graduate School Fellowship and a Walter P. Murphy Fellowship. He is a recent recipient of the NSF CAREER award (2012), as well as the ISU award for Early Achievement in Teaching (2012) and the ECpE Department's Warren B. Boast undergraduate teaching award (2009, 2011).

Phillip H. Jones received his BS in 1999 and MS in 2002 in Electrical Engineering from the University of Illinois at Urbana-Champaign, and his PhD in 2008 in Computer Engineering from Washington University in St. Louis. Currently, he is an Assistant Professor in the Department of Electrical and Computer Engineering at Iowa State University, Ames, where he has been since 2008. His research interests are in adaptive computing systems, reconfigurable hardware, embedded systems, and hardware architectures for application-specific acceleration.

This paper is a revised and expanded version of a paper entitled 'Improving system predictability and performance via hardware accelerated data structures' presented at Proceedings of the International Conference on Computational Science (ICCS), 2012, Omaha, Nebraska, USA, 4–6 June.

1 Introduction

Deploying increasing amounts of computation into smaller form factor devices is required to keep pace with the ever increasing needs of real-time and embedded system applications. The area of micro unmanned aerial vehicles (UAVs) is an example of where such need exists. The size of these vehicles has rapidly decreased, while the capabilities users wish to deploy continue to explode. As recently as June 2011, the *New York Times* published several articles on the cutting-edge work being pursued by Wright Patterson Air Force Base to develop micro-drones to aid soldiers on the battlefield (Bumiller and Shanker, 2011). In February of 2011, the DARPA funded nano air vehicle (NAV) program demonstrated a humming bird form-factor UAV weighing less than 20 grams (e.g., less than an AA battery) (DARPA, 2011; Grossman et al., 2011) with video streaming capabilities. These real-time and embedded applications can no longer rely on manufacturing advances to provide computing performance at Moore's law rates, owing to transistors approaching atomic scales and thermal constraints (ITRS, 2009). Thus, more efficient use of the transistors available is needed. For example, use of application specific hardware has showed promise in accelerating various application domains, from cryptography (Eberle et al., 2008; Ors et al., 2008) to numerical simulation (Rahmouni et al., 2013) to control systems (Muller et al., 2013).

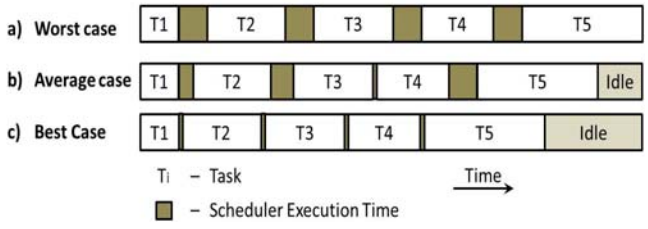
We assert that the boundaries of software and hardware must be reexamined and we believe a fruitful realm for research is the hardware-software co-design of functionality that has been traditionally implemented in software. Such a co-design is needed to balance the cost of dedicating limited silicon resources for high-performance fixed hardware functionality, with the flexibility and scalability offered by software. Additionally, we claim seamless migration between software and hardware implemented functionality is required to allow systems to adapt to the dynamic needs

of applications. In this paper we examine a hybrid architecture for priority queue management and evaluate this architecture within a real-time scheduling context. The following motivates the importance of low processing overhead and timing predictably to a real-time scheduler's performance.

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 (Park et al., 2001; Adomat et al., 1996).

The extent to which a scheduler can ideally implement a given scheduling paradigm [e.g., earliest deadline first (EDF), rate monotonic (RM)], 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 summarised 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. Reducing the scheduler's timing sensitivity to such factors can help increase deterministic behaviour, which in turn allows the scheduler to better model a given scheduling paradigm.

Figure 1 In order to allow analytical analysis of schedule feasibility, worst-case execution time (WCET) typically needs to be assumed (see online version for colours)



Note: Thus, scheduler execution time variations that cause large differences between WCET and typical case execution time reduce utilisation of system computing resources.

Figure 1 illustrates how the variation in scheduler overhead affects processor utilisation. To ensure that tasks meet their deadlines, the scheduler's worst-case execution times are often overestimated. This can cause a system to be underutilised 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 interrupt processing) to hardware logic. The expected results of our efforts are increased CPU utilisation, better system predictability, finer schedule and timing resolution.

1.1 Contributions

The primary contributions of this paper are

1 a hardware accelerated binary min heap that supports enqueue and peek operations in O(1) time, returns the top-priority element in O(1) time, and completes a dequeue operation in $O(\log n)$ time

2 a scalable hardware-software priority queue architecture that enables fast and low-overhead transitions of queue management between hardware and hybrid modes of operation

3 a hybrid scheduler architecture that reduces scheduling overhead and improves predictability.

1.2 Organisation

The reminder of this paper is organised as follows. Section 2 describes the hardware-software priority queue architecture and implementation details. Section 3 describes the hardware scheduler architecture, which uses our priority queue design. The evaluation methodology and results are discussed in Sections 4 and 5. Section 6 presents related work on hardware accelerated priority queues and schedulers. Conclusions and future work are presented in Section 7.

2 Hybrid priority queue architecture

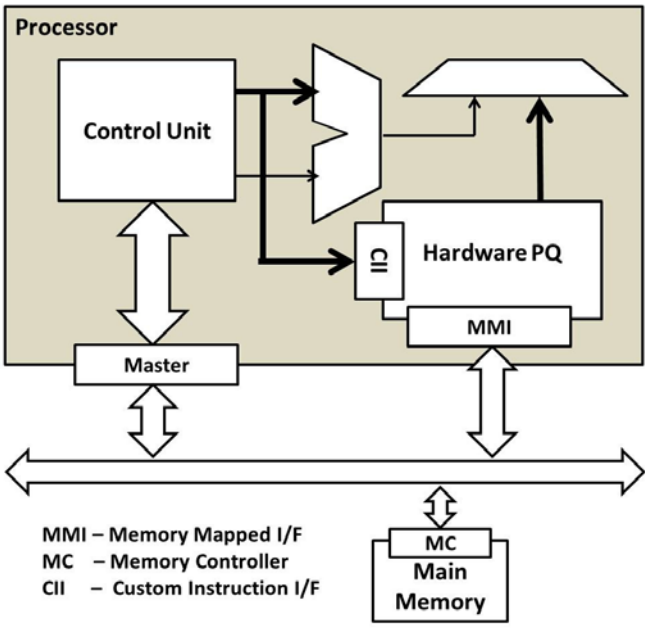
Priority queues are commonly implemented using a binary heap data structure, which supports enqueue and dequeue operations in $O(\log n)$ time. A binary heap is constrained by the heap property, where the priority of each node is always less than or equal to its parent. In a binary min heap, lower key-value corresponds to higher priority and the root node has the highest priority (lowest key value). A binary heap can be 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 hybrid priority queue architecture that includes the hardware implementation of a conventional binary min heap (lower key value corresponds to higher priority), which can be managed in hardware and/or software. A binary heap could be stored compactly when compared to skip list, binomial heap and Fibonacci heap, without requiring additional space for pointers. Since the memory available in hardware (on-chip memory) is limited, the priority queue was implemented as a binary heap to better utilise the available resources. The priority queue operates in hardware mode when the queue size is less than a hardware limit threshold. When managed in hardware, the

priority queue 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 (lowest key value) element can be returned immediately, allowing the dequeue operation to overlap its execution with the primary processor. Software issues custom instructions to initiate hardware-implemented enqueue and dequeue operations.

Once the priority queue size exceeds hardware limits, excess elements are stored in the system's main memory and managed by both hardware and software. Elements of the priority queue that are managed by hardware are memory mapped, providing software with direct access to these elements that are stored in a priority-queue-structured on-chip memory. Figure 2 illustrates this architecture. Memory mapping the priority-queue-structured on-chip memory additionally allows rarely used priority queue operations (e.g., delete element and decrease key) to be easily implemented in software, thus reducing the complexity of hardware control logic.

Figure 2 A high level block diagram of the hardware-base priority queue interface (see online version for colours)



2.1 Hardware priority queue

A high level architecture diagram for the priority queue is shown in Figure 3. Central to the priority queue is the queue manager, which provides the necessary interface to the CPU and executes operations on the queue. Elements in each level of the binary heap are stored in separate on-chip memories called block rams (BRAMs) to enable parallel access to elements, similar to Bhagwan and Lin (2000) and Ioannou and Katevenis (2007). The address decoder generates addresses and control signals for the BRAM blocks. Queue operations in hardware mode are explained in detail below, using a min-heap example, where a lower key value corresponds to a higher priority.

2.1.1 Enqueue

Enqueue operations in a software binary heap are accomplished by inserting the new element at the bottom of the heap and performing compare-swap operations with successive parents until the priority of the new element is less than its parent. In software, the worst-case behaviour of this operation 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 [i.e., $O(\log n)$ time].

However, our hardware implementation can perform this operation in $O(1)$ time. We first calculate the path from the next vacant leaf node to the root. The index, i, of this leaf node is always one more than the current size of the queue, and each ancestor of this leaf node can be computed in parallel using a closed form equation (e.g., k^{th} parent is located at index $i/2^k$) in hardware. 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 Moon et al. (1997). Each level of the heap is mapped to an enqueue cell, which consists of a comparator, multiplexer 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 the 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 3 The hardware priority queue architecture (see online version for colours)

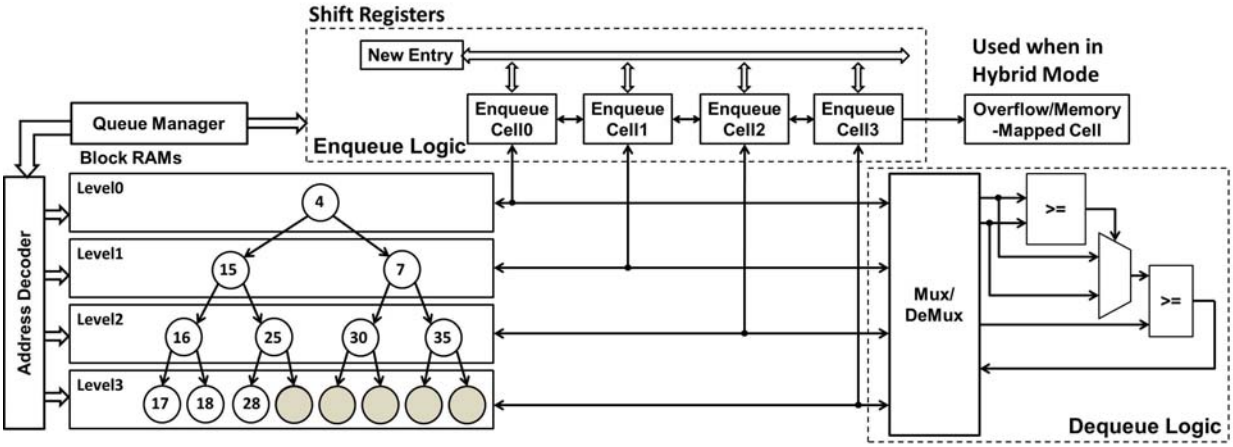


Figure 4 Steps of enqueue operation in hardware mode, (a) elements in the insertion path are loaded to enqueue cells (b) sorted insert of the new element to the enqueue cell array (c) elements in the enqueue cell array are stored back to the heap (see online version for colours)

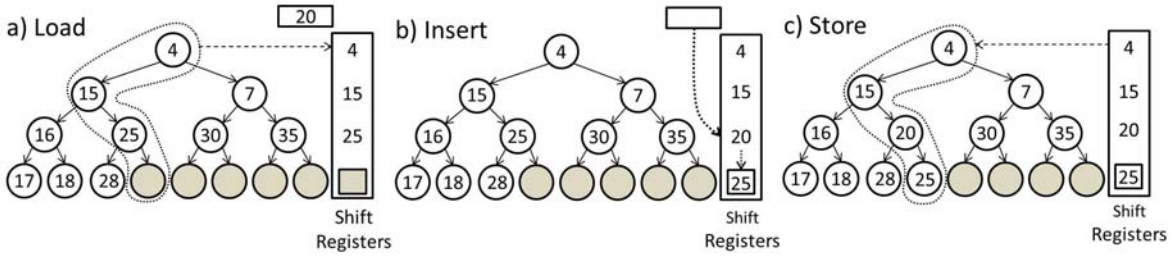
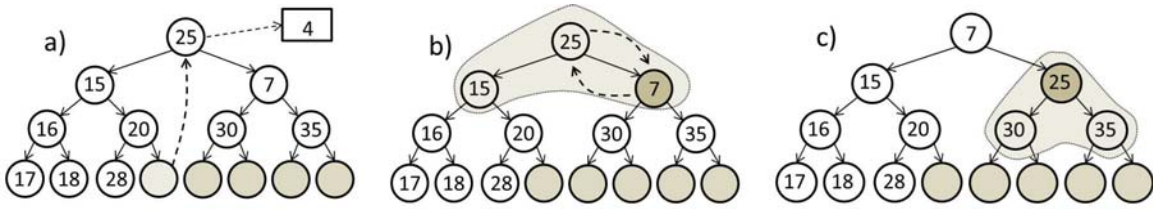


Figure 5 Steps of dequeue operation in hardware mode, (a) the root element is removed by replacing it with last element of the queue (b) new root is swapped with highest priority child (c) no more swap operations as the heap property is restored (see online version for colours)



Note: In worst case there will be log(n) swap operations.

2.1.2 Dequeue

Figure 5 illustrates an example of a dequeue operation in hardware mode. The dequeue operation can be divided into two stages: 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 first removed by replacing it with the last element of the queue to keep the heap balanced. The new root element is then compared with its highest priority child and is swapped if its priority is less than that of its child. This operation is repeated until the priority of the new root element is greater than that of its children.

Note that the root element is returned immediately to the processor before restoring the heap property. The processor is not required to wait for the operation to complete, as the heap property of the queue is restored in hardware which executes in parallel to the CPU. Back-to-back dequeue operations would cause the processor to wait for the first operation to be completed in hardware before getting the result of the second request. Hence, the worst case execution time of a dequeue operation is $O(\log n)$.

2.1.3 Decrease-key and delete

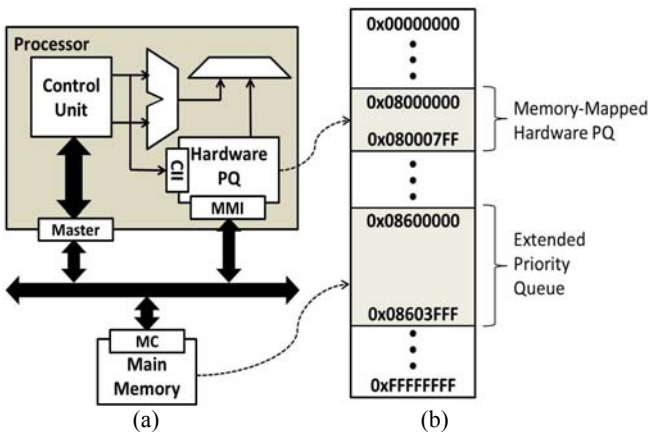
The decrease-key operation decreases the priority of a given queue element, and the delete operation removes a specified element from the queue. Supporting these rarely used operations in hardware adds considerable complexity to the hardware's control logic. To avoid this complexity, these operations have been implemented in software. Software accesses the hardware priority queue elements via a memory mapped interface as if they resided in main memory.

2.2 Hybrid priority queue management

The size of the hardware priority queue is limited by the available on-chip memory resources of the device. Gracefully handling size overflow situations allows the use of hardware data structures for a wider range of applications. We achieve this by extending the heap array to off-chip memory (i.e., main memory) and managing the queue in both hardware and software. In hybrid mode, the enqueue and dequeue operations are executed in two stages. The hardware executes a part of the queue operation in the first stage, and then control is returned to software, which completes the rest of the operation.

A memory mapped interface, shown in Figure 6(a), provides software access to on-chip priority queue elements as if they were resident in main memory. Since the address space of memory mapped hardware and the extended priority queue will typically not be part of the same continuous memory block, as shown in Figure 6(b). The queue algorithm needs to be modified accordingly to access the correct address depending on the array index of the element. The combination of memory mapping the hardware-base priority queue and implementing small modification to the queue algorithm enables our hybrid approach to have fast and low overhead transitioning between hardware and software management. The priority queue operations in hybrid mode are explained in detail below.

Figure 6 (a) Memory mapped interface provides access to priority queue elements stored in BRAM (b) Virtual address space showing extended priority queue (see online version for colours)



2.2.1 Enqueue

Figure 7 presents an example of the enqueue operation in hybrid mode. In the first stage of an enqueue operation, the new element is inserted into the hardware priority queue, which forms the top portion of the queue. This is similar to the hardware enqueue operation as explained in Section 2.1.1. Since we only go into hybrid mode when the queue size exceeds hardware limits, the lowest priority element in the hardware insertion path must be moved to the overflow buffer shown in Figure 3. This first stage is performed in constant time as explained in Section 2.1.1. Control is then returned to software. The overflow buffer is available to software through a memory mapped interface. In the second stage of the enqueue operation, the element in the overflow buffer is copied to the bottom of the extended queue and compare-swap operations are performed with successive parents until the heap property is restored. This stage is similar to the software enqueue operation and only the extended part of the queue (stored in main memory) is modified by software. The software implementation of enqueue operation is outlined in Algorithm 1.

Algorithm 1 Pseudocode of hybrid priority queue's enqueue operation

```
1:  procedure HYBRiD_PQ_ENQUEUE(queue, elem)
2:      if Queue = Full then
3:          throwexception
4:      end if
5:      Hardware_pq_enqueue(elem)
6:      queue.size + +
7:      if queue.size > queue.hwlimit then
8:          index = queue.size
9:          Copy overflown hardware element to the end of
            software queue.
10:         queue.data[index] = overflow_cell
11:         while index > queue.hw_limit do
12:             if queue.data[index]
                <queue.data[parent(index)} then
13:                 swap_queue_data(queue, index,
                    parent(index))
14:                 index = parent(index)
15:             end if
16:         end while
17:     end if
18: end procedure
```

Figure 7 Steps of enqueue operation in hybrid mode, in this example we assume that the first 3 levels of the heap are managed in hardware, (a) hardware elements in the insertion path are loaded to enqueue cells (b) sorted insert of the new element and the lowest priority element is moved to the overflow buffer (c) hardware stores back the elements in enqueue cells and the overflow buffer element is moved to the bottom of the queue by software (d) software performs compare-swap operation to restore heap property (see online version for colours)

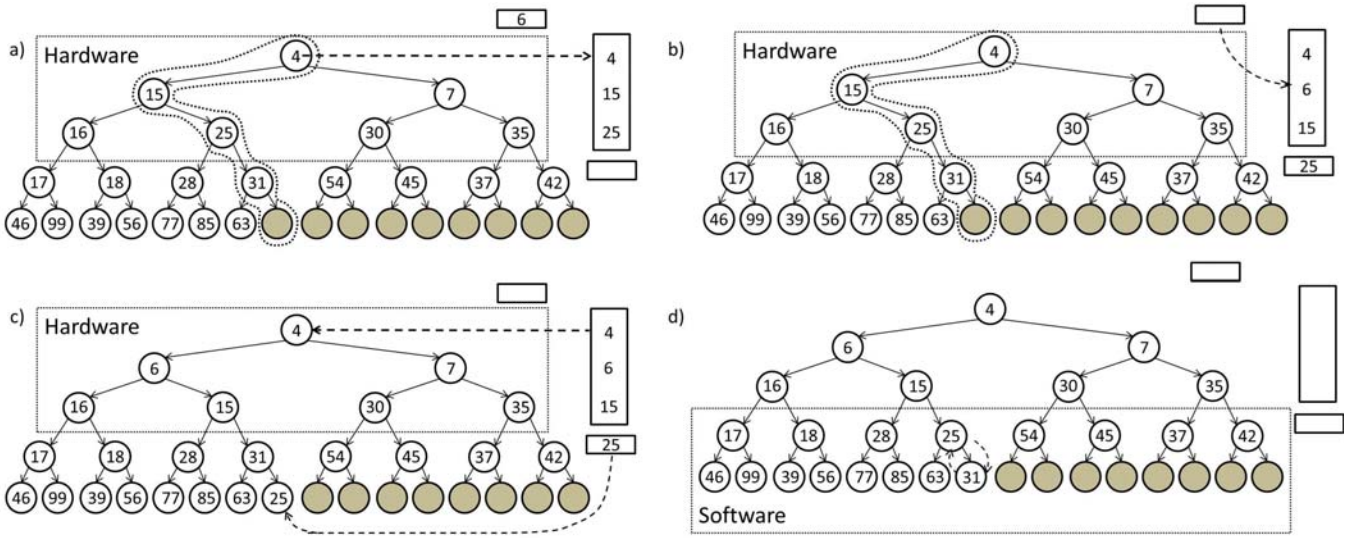
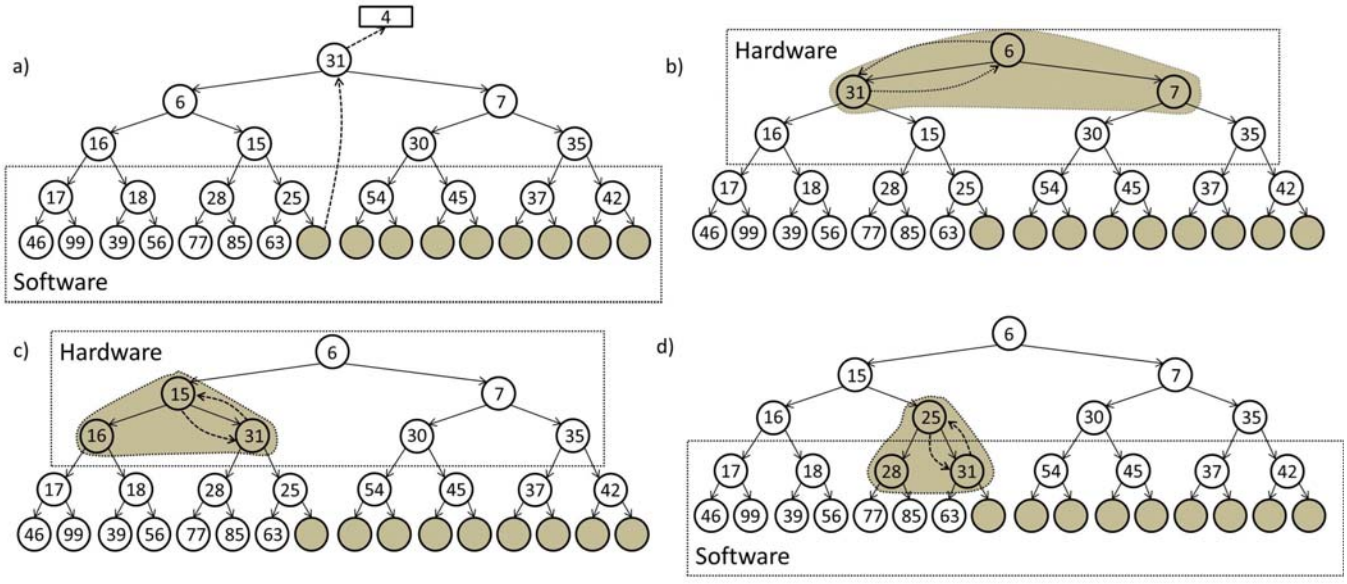


Figure 8 Steps of dequeue operation in hybrid mode, in this example we assume that the first three levels of the heap are managed in hardware, (a) the root element is removed by replacing it with the last element of the queue by software (b) the heap property is restored by swapping the new root (31) with highest priority child (c) hardware completes dequeue operation and returns the position of new root(31) (d) software continues restoring the heap property from the position returned (see online version for colours)



2.2.2 Dequeue

Figure 8 provides an example of the dequeue operation in hybrid mode. In the first stage of a dequeue operation, the root element of the queue is removed by replacing it with the last element of the queue. This operation should be performed by software, since the last element of the queue resides in main memory. The hardware dequeue operation is then initiated through a custom instruction, which restores the heap property of the hardware portion of the queue as explained in Section 2.1.2. The custom instruction when completed returns the position of the newly inserted element, which can be accessed by software through memory mapped interface. The software then continues restoring the heap property starting from the position returned. The software implementation of dequeue operation is outlined in Algorithm 2.

Comparing our approach with the related work reported in Section 6, our approach scales nicely without requiring complex hardware control logic to manage pipelining. Our hardware-software co-design approach overcomes the size

limitations of hardware, enabling the support of arbitrarily large priority queues.

3 Hardware scheduler

3.1 Overview

As an application of the priority queue described above, we propose a hardware-software scheduler architecture designed to reduce the time-tick interrupt processing and scheduling overhead of a system. In addition, our hybrid architecture increases the timing determinism of the scheduler operations. The instruction set architecture of a processor was extended to support a set of custom instructions to communicate with the scheduler. The hardware scheduler executes the scheduling algorithm and returns control to the processor along with the next task to execute. Software then performs context switching before executing the next task.

Algorithm 2 Pseudocode of hybrid priority queue's dequeue operation

```
1:  procedure HYBRiD_PQ_DEQUEUE(queue)
2:      if Queue = Empty then
3:          throw exception
4:      end if
5:      result = queue.top;
6:      if queue.size < queue.hw_limit then
7:          hardware_pq_dequeue()
8:      else
9:          Replace root with last element of heap array.
10:         queue.data[0] = queue.data[size]
11:         Execute hardware dequeue and return position of
            newly inserted element.
12:         new_index = hardware_pq_dequeue()
13:         Continue heap restoration in software from the
            position returned.
14:         Restore_sw_heap(new_index)
15:     end if
16:     queue.size – –;
17: end procedure
```

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 chooses to continue executing the current task or preempts it to run a higher priority task. The following describes the functionality of the key components of the hardware accelerated scheduler.

3.2 Architecture

A high level block diagram of the hardware scheduler is shown in Figure 9.

3.2.1 Controller

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 (ready queue and sleep queue).

3.2.2 Timer unit

The timer unit keeps track of the 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 run time.

3.2.3 CPU interface

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 arbitration timing dependencies for data transfer. Basic scheduler operations such as run, configure, add task, and preempt task are supported.

3.2.4 Task queues

At the core of the scheduler are the task queues, which are implemented as priority queues. The ready queue stores active tasks based on their priority. The sleep queue stores inactive tasks until their activation time. The task with the earliest activation time is located at the front of the sleep queue.

3.3 Modes of operation

The scheduler is designed to operate in either hardware or hybrid mode, depending on the size of the hardware priority queues and the number of tasks in the system. Once the number of tasks exceeds the hardware limit, the queues extend to off-chip memory (i.e., main memory) and the scheduler starts operating in hybrid mode. In hybrid mode the scheduling algorithm is executed in software and the hybrid priority queues described in Section 2 are used to accelerate scheduler operations. This transition involves stalling the hardware scheduler through a co-processor call (custom instruction) and calling the software scheduler function. As the elements stored in the on-chip priority queues can be accessed by software via a memory mapped interface, it avoids the need to copy data between hardware and software memory when the scheduler changes modes. The proposed scheduler architecture scales to support an arbitrarily large number of tasks.

4 Evaluation methodology

4.1 Platform

The hybrid priority queue and the scheduler were deployed and evaluated on the re-configurable autonomous vehicle

infrastructure (RAVI) board, an in-house developed FPGA prototyping platform. 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 co-design space for designing system architectures that best fit an application's requirements. The portions of the RAVI board used for our experiments included the Cyclone III FPGA, the on-board DDR DRAM and the UART. The FPGA was used to implement the NIOS-II (Altera's soft-processor), the DDR stored software that was executed on the NIOS-II, and the UART supported data collection. A pictorial description of the setup is shown in Figure 10.

4.2 Architecture configuration

The priority queue and the scheduler were implemented as an extension to the instruction set architecture (using custom instructions) of a Nios II embedded processor running at 50 MHz on an Altera Cyclone III FPGA. The priority queue supported up to 255 elements in hardware mode and up an arbitrarily large number of elements in hybrid mode of operation. For our evaluation we limited the queue size to 8,192 elements. A binary heap-based priority queue implemented in software was used as a baseline to compare against the performance of our hybrid priority queue.

The scheduler can support up to 255 tasks when managed in hardware, and up to an arbitrarily large number of tasks when in hybrid mode. For our evaluation we limited the task set size to 2,048, which is sufficient to support a vast majority of embedded systems. The scheduler can be configured to use EDF or a fixed priority-based scheduling algorithm such as rate monotonic scheduling (RMS). The scheduler overhead was also measured using different timer-tick resolutions (0.1 ms, 1 ms, 10 ms), which is used to generate periodic interrupts for the scheduler. A software test bench was built to accurately measure the overhead of the scheduler for different task sets and timer resolutions. Hardware-based performance counters, supported by the NIOS II processor provided a relatively unobtrusive mechanism to profile software programs including interrupt service routines in real-time. An EDF (Liu and Layland, 1973) scheduler was deployed to measure the impact of running a dynamic scheduling algorithm on the processor. In EDF scheduling, task priorities are assigned based on the absolute deadline of the current request. At any given time, the task with the nearest deadline will be assigned the highest priority and executed. A software EDF scheduler implementation was used as a baseline to compare against our hybrid implementation.

Figure 9 A high level architecture diagram of the hardware scheduler along with the custom instruction interface (see online version for colours)

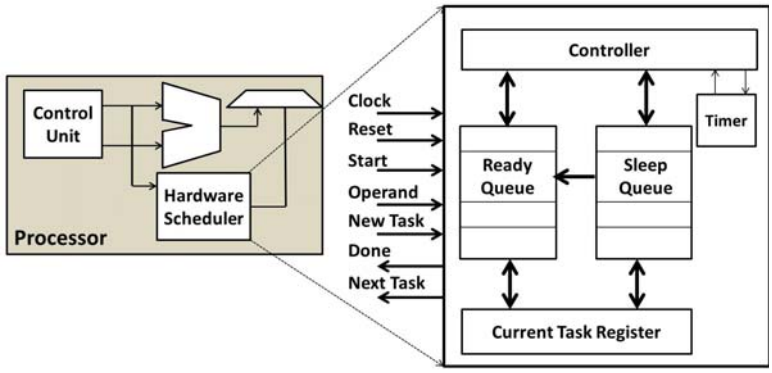
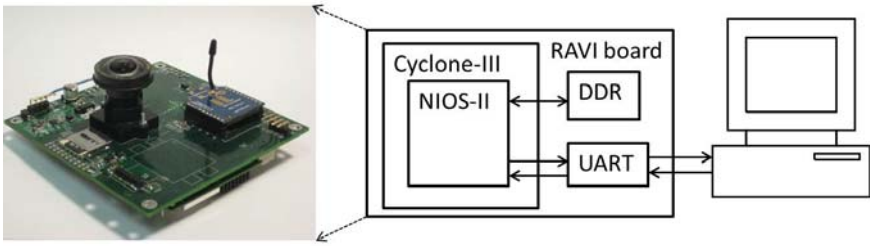


Figure 10 FPGA-based evaluation platform (see online version for colours)



4.3 Workload and metrics

The performance of the priority queue was evaluated using the classic hold model (Vaucher and Duval, 1975); Jones, 1986), where a priority queue of a given size is initialised and hold operations (dequeue followed by enqueue) are performed repeatedly on the queue. The size of the queue remains constant for the whole duration of the experiment. The access time measured by the hold model is dependent on the initial queue size and priority increment distribution. For our evaluation we used exponential, uniform, bimodal and triangular distributions, similar to those used in Vaucher and Duval (1975) and Ronngren and Ayani (1997). The transient behaviour of the priority queue is measured using the up/down model (Ronngren et al., 1991), where the queue is initialised to a given size by series of enqueue operation and then emptied by series of dequeue operation.

A set of periodic tasks with randomly generated parameters (i.e., task execution time and period) was used to evaluate the performance of the EDF scheduler. The relative deadline of the tasks were assumed to be equal to their period. The number of tasks in the task set were varied, keeping the utilisation factor constant at 80%. The metrics used to evaluate our scheduler were:

- scheduler overhead: time spent executing the scheduling algorithm
- timer-tick overhead: time taken to service the periodic timer interrupt
- predictability: variation in the execution time of individual scheduler invocations.

5 Results and analysis

This section presents the results of our hybrid priority queue versus software priority queue evaluation experiments. A discussion is then given on the results of our hybrid and hardware scheduler evaluation experiments.

5.1 Priority queue

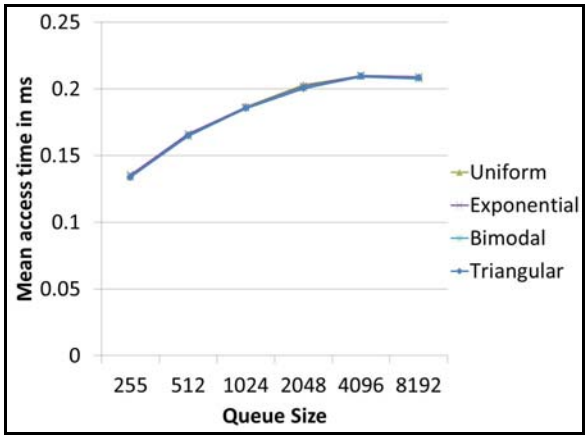
5.1.1 Mean access time

The mean access times of the hybrid and software priority queues measured using classic hold and up/down experiments are shown in Figures 11 and 12. The hybrid priority queue is fully managed in hardware when the queue size is 255 or less. The results show that the hybrid queue is six times faster than the software queue when the queue size is 255. The hybrid priority queue extends to software memory when the queue size exceeds 255 elements and the fraction of total work done in hardware decreases as more levels of heap are stored in software memory. Hence, the difference in performance between the hybrid and software priority queue decreases as the size of the queue increases. Even when the queue contains 8,192 elements, the hybrid priority queue performs close to 30% better than software priority queue. The performance of the hybrid and software priority queue is not very sensitive to priority increment distributions.

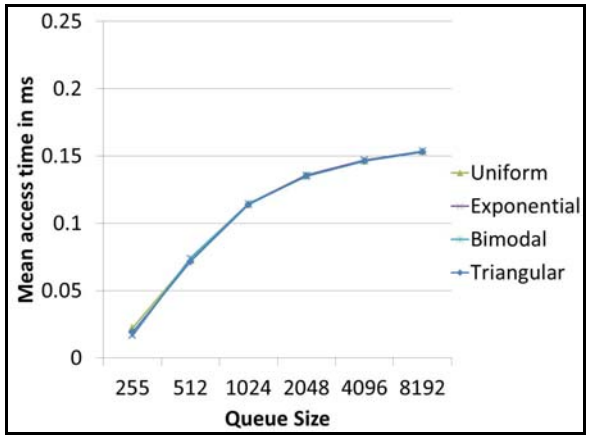
5.1.2 Resource utilisation and scalability

We implemented our hardware priority queue design on an Altera Cyclone III (EP3C25) FPGA. The resource utilisation of the priority queue for different queue lengths is shown in Table 1. Each priority queue element is 64 bits wide, with a 32 bit priority value. The amount of combinational logic required increases *logarithmically* with the size of priority queue. Since the number of elements doubles with each additional level, the combinational logic scales logarithmically with queue size. The device contains 66 M9K memory blocks, which can be used as on chip memory. Each M9K block can hold 8,192 memory bits with a maximum data port width of 36. Since each level of the heap is stored in a BRAM with a 64 bit wide port, a minimum of 2 M9K blocks are used per level. The BRAM usage can be optimised by moving the first 5 levels of the heap to memory mapped registers. We also implemented the shift-register and systolic array-based priority queue architectures described in Moon et al. (1997). The resource utilisation of both architectures are shown in Table 2. These architectures use distributed memory instead of BRAMs to store queue elements. Figure 13 shows that our queue architecture scales well for large queues, as compared to shift-register and systolic array-based architectures (Moon et al., 1997) in which the combinational logic required increases linearly with queue size.

Figure 11 Performance comparison between the software and hybrid implementation of a priority queue, (a) software priority queue (b) hybrid priority queue (see online version for colours)



(a)



(b)

Note: Evaluated using the classic hold model, for four different priority increment distributions.

Table 1 FPGA resource utilisation of the proposed priority queue design for different queue sizes

Size	*Resources*[1]			
	Look-up tables (LUTs)	*Flip-flops*	*Memory (bits)*	*BRAMs*
31	1,411 (5.73%)	906 (3.68%)	1,920 (0.32%)	8 (12.12%)
63	1,996 (8.1%)	1,048 (4.25%)	3,968 (0.65%)	10 (15.15%)
127	2,561 (10.4%)	1,182 (4.8%)	8,064 (1.325%)	12 (18.18%)
255	3,161 (12.84%)	1,330 (5.4%)	16,256 (2.67%)	14 (21.21%)

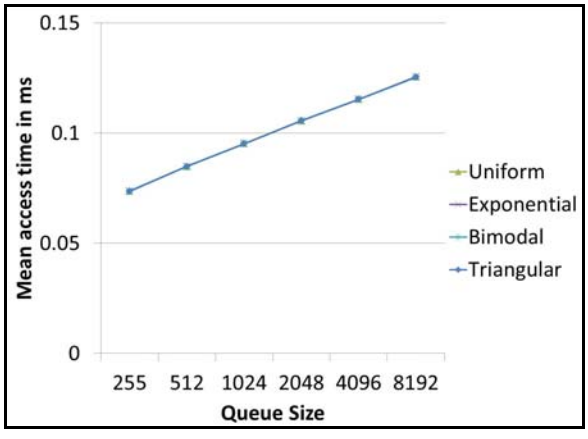
Note: [1]Altera Cylone III FPGA contains: 24,624 LUTs, 24,624 flip-flops and 66 BRAMs.

Table 2 FPGA resource utilisation of shift register and systolic array-based priority queue architectures (Moon et al., 1997) in comparison with proposed priority queue design

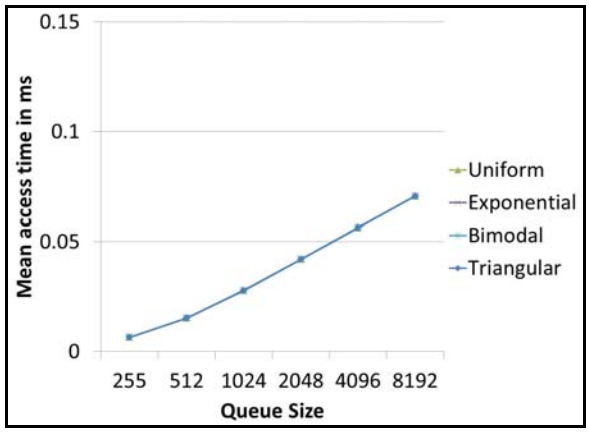
Size	*Shift register*		*Systolic array*		*Proposed design*	
	LUTs	*Flip-flops*	*LUTs*	*Flip-flops*	*LUTs*	*Flip-flops*
31	4,995 (20.29%)	2,077 (8.43%)	8,560 (34.76%)	3,999 (16.24%)	1,411 (5.73%)	906 (3.68%)
63	10,275 (41.73%)	4,221 (17.14%)	17,520 (71.15%)	8,127 (33.00%)	1,996 (8.1%)	1,048 (4.25%)
127	20,835 (84.61%)	8,509 (34.56%)	–	–	2,561 (10.4%)	1,182 (4.8%)
255	–	–	–	–	3,161 (12.84%)	1,330 (5.4%)

Note: – Configurations for which the priority queue resources do not fit in Altera Cyclone III FPGA.

Figure 12 Performance comparison between the software and hybrid implementation of a priority queue, (a) software priority queue (b) hybrid priority queue (see online version for colours)



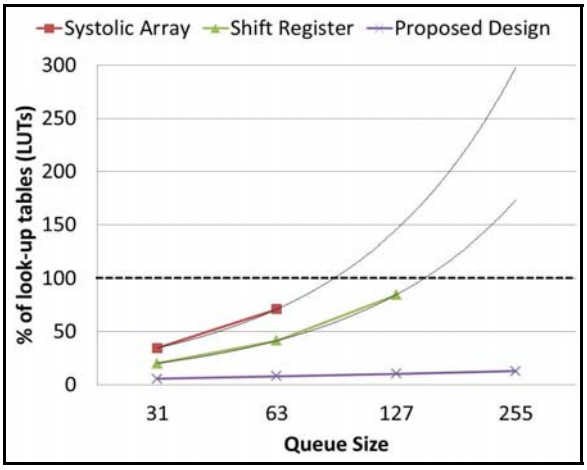
(a)



(b)

Note: Evaluated using the up/down model, for four different priority increment distributions.

Figure 13 Comparing FPGA look-up table utilisation of the proposed priority queue design against shift register and systolic array-based priority queue architectures (Moon et al., 1997) for different queue sizes (see online version for colours)



Note: Flip-flop utilisation also shows a similar trend.

5.2 Scheduler

For our analysis we have considered the following three configurations of an EDF scheduler.

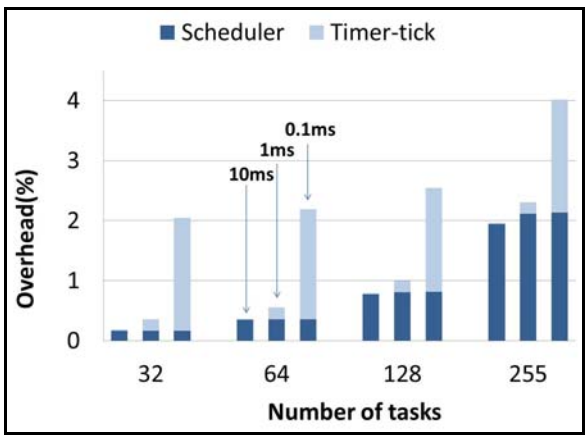
- Software scheduler: used as the baseline for evaluating our hybrid and hardware scheduler. Evaluated for up to 2,048 tasks.
- Hardware scheduler: executes scheduling algorithm, manages task queues, and supports up to 255 tasks in hardware.
- Hybrid scheduler: the task queues of the software scheduler is replaced by our hybrid priority queue to accelerate scheduler operations. Evaluated for up to 2,048 tasks.

5.2.1 Scheduler overhead

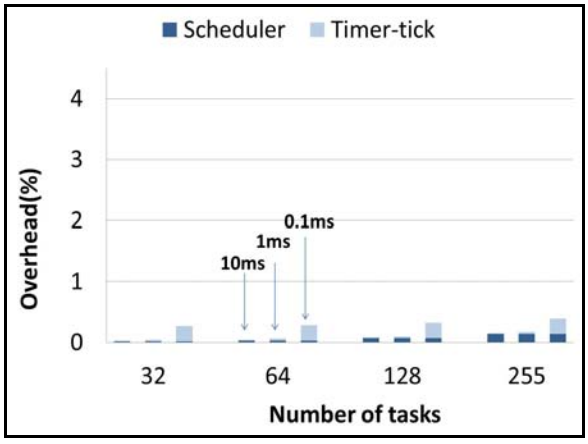
The overhead of the scheduler was measured for different sets of tasks and timer tick resolutions. Figure 14(a) shows the percentage overhead of software scheduler. The software scheduler overhead increases with the number of tasks and the timer-tick resolution. Most of this overhead results from time tick processing, where the scheduler periodically processes interrupt requests to check for new tasks and managing the task queues. This time-tick processing has been a limiting factor for implementing dynamic priority-based scheduling algorithms in embedded real time systems (Park et al., 2001; Adomat et al., 1996), since finer granularity time ticks lead to closer to ideal implementation of such schedulers.

Figure 14(b) shows the scheduling overhead when the hardware scheduler is used. The results show that when the timer tick resolution is set to 0.1 ms and with 255 tasks, the scheduler overhead is less than 0.4%. This is a 90% reduction in scheduler overhead as compared to the software implementation. 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 now replaced by a custom instruction call to obtain the next task for execution or to preempt the current task. The overhead of managing the task queues in software is removed, as the scheduler runs in parallel to the processor and hardware priority queues are used to accelerate task queue management. The time tick processing overhead is reduced considerably as the software interrupt service routine just needs to execute a single instruction to check the availability of a higher priority task in the hardware scheduler.

Figure 14 Performance of the software scheduler compared with hardware scheduler for task sizes less than or equal to 255, (a) software scheduler (b) hardware scheduler (see online version for colours)

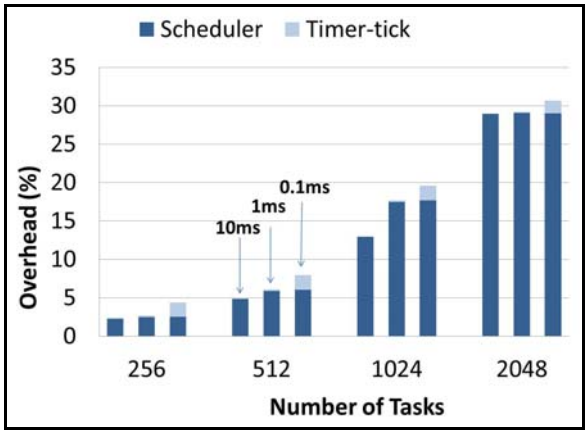


(a)

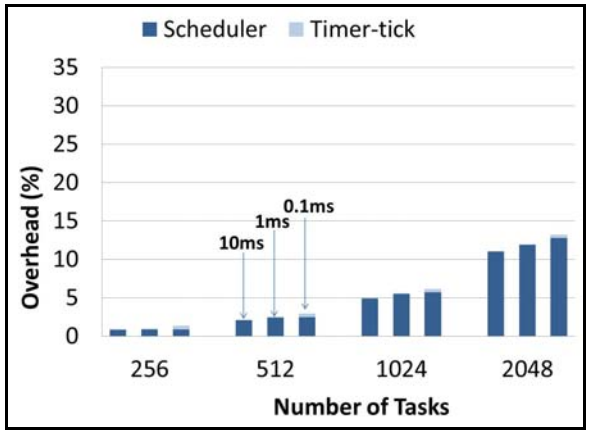


(b)

Figure 15 Performance of software scheduler compared with hybrid scheduler for task sizes greater than 255, (a) software scheduler (b) hardware scheduler (see online version for colours)

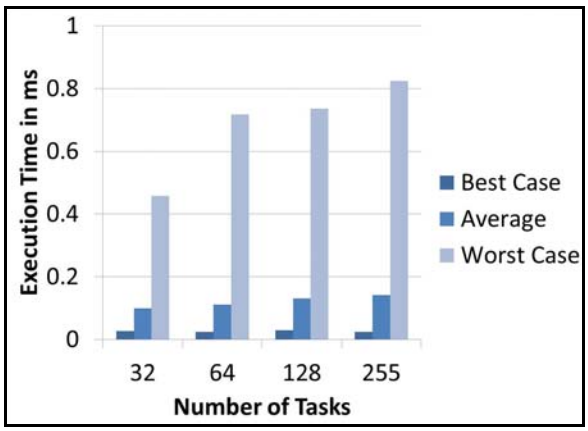


(a)

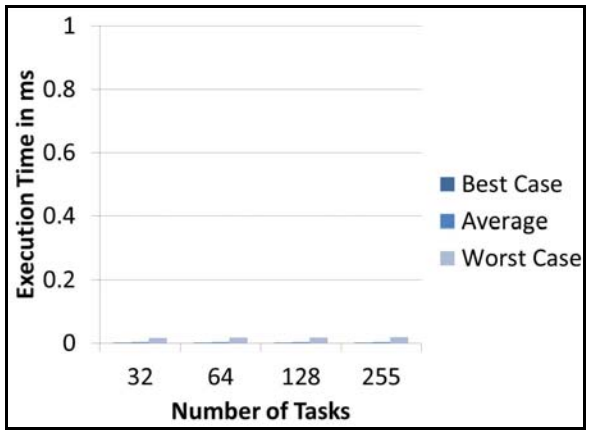


(b)

Figure 16 Variation in execution times of software and hardware scheduler, (a) software scheduler (b) hardware scheduler (see online version for colours)



(a)



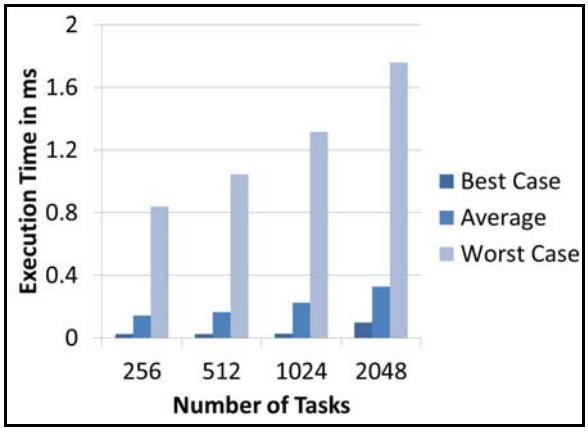
(b)

Once the number of tasks exceeds 255, our scheduler executes in hybrid mode where the scheduling algorithm runs in software and queue operations are accelerated using our hybrid priority queues. The switching between hardware and hybrid scheduler mode is quick and has little or no overhead in part due to the hardware queues being memory mapped. The overhead of the scheduler in hybrid mode is 50% less than the software scheduler overhead as seen in Figure 15.

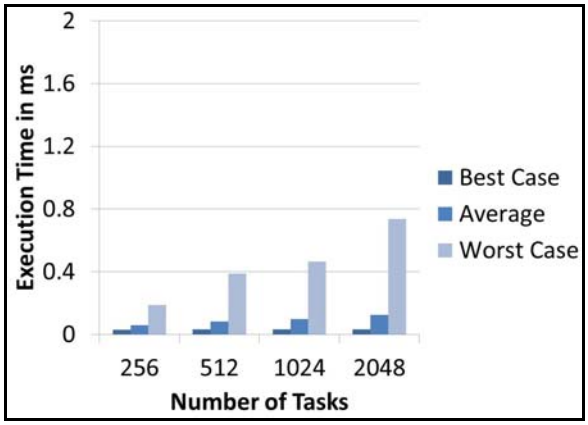
5.2.2 Predictability

The predictability of the scheduler can be measured as the variation in the execution time of a single call to the scheduler. The variation in execution times of the hardware and software scheduler is shown in Figure 16. The difference between the best case and worst case execution time of the software scheduler is 50 times larger then the hardware implementation as shown in Figure 16. This variation for the software implementation is due to system factors such as changes in task-set composition, cache misses, etc. The processing time of the software priority queues (task queues) varies, as it depends on the current queue size and task parameters. These variations can make the scheduler a significant source of non-determinism in real-time systems. Since the system must be designed for worst case behaviour to ensure task deadlines are met, increases in execution time variation reduces CPU task utilisation (i.e., CPU becomes underutilised). On the other hand, the execution times of the hardware scheduler show more deterministic behaviour with very little variation. Migrating time-tick processing to hardware and the use of hardware accelerated priority queues results in tighter worst-case execution time bounds for the scheduler. This in turn leads to higher CPU task utilisation. Figure 17 shows the variation in execution time of the hybrid scheduler in comparison with the software scheduler. The use of hybrid priority queues in the software scheduler reduces the variation in the scheduler execution time by more than 50% as shown in Figure 17.

Figure 17 Variation in execution times of software and hybrid scheduler, a) software scheduler (b) hardware scheduler (see online version for colours)



(a)



(b)

6 Related work

6.1 Hardware priority queues

Many hardware priority queue architectures have been implemented in the past, most of them in the realm of real-time networks for packet scheduling (Moon et al., 1997; Bhagwan and Lin, 2000; Ioannou and Katevenis, 2007). Moon et al. (1997) compared four scalable priority queue architectures: first-in-first-out, 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 architectures expensive to scale in terms of hardware resources.

Bhagwan and Lin (2000) 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 Ioannou and Katevenis (2007) 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.

The size of the priority queues discussed above is limited by the availability of on-chip memory. A hybrid priority queue system (HPQS) was proposed in Zhuang and Pande (2006), where both SRAM and DRAM was used to store large priority queues used in high speed network devices. A java-based hardware-software priority queue was proposed in Chandra and Sinnen (2010), where a shift-register-based priority queue (Moon et al., 1997) was extended by appending a software binary heap. Bloom et al. (2012) presented an exception-based mechanism for handling overflows in hardware priority queue, where additional data is moved to secondary storage by the exception handler.

6.2 Hardware schedulers

Several architectures (Adomat et al., 1996; Burleson et al., 1999; Saez et al., 1999; Kuacharoen et al., 2003; Gupta et al., 2010; Kohout et al., 2003) have been proposed to improve the performance of schedulers using hardware accelerators. Most schedulers implement some kind of priority-based scheduling algorithm that requires a priority queue to sort the tasks based on their priority. A real time kernel called FASTHARD has been implemented in hardware (Adomat et al., 1996). The scheduler of FASTHARD can handle 256 tasks and eight priority levels. The Spring scheduling coprocessor (Burleson et al., 1999) was built to accelerate scheduling algorithms used in the Spring kernel (Stankovic and Ramamritham, 1991), which was used to perform feasibility analysis of the schedule. Kuacharoen et al. (2003) implemented a configurable hardware scheduler that provided support for three scheduling disciplines, configurable during runtime. A slack stealing scheduling algorithm was implemented in hardware (Saez et al., 1999) to support scheduling of tasks (periodic and aperiodic) and to reduce scheduling overhead. Nakano et al. (1995) implemented most of the/xITRON kernel functionality including tasks scheduling in a co-processor called STRON-1 which reduced the kernel overhead. A hardware scheduler for multiprocessor system on chip is presented in Gupta et al. (2010), 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 Kohout et al. (2003). The RTM supports static priority-based scheduling and is implemented as an on-chip peripheral that communicates with the processor though a memory mapped interface. The SERRA run-time scheduler synthesis and analysis tool was presented in Mooney and Micheli (1997). The tool automatically generated a run-time hardware-software scheduler from system level specification. A hardware-software kernel was presented in Morton and Loucks (2004), which implemented a scheduling co-processor running EDF scheduling algorithm.

A hardware real-time scheduler coprocessor (HRTSC) architecture for NIOS II processor was described in Varela et al. (2012), which could be configured to run any priority-based scheduling discipline.

The hardware priority queues described above use on-chip memory to store data, which limits the size of the queue due to resource constraints of the device. In our hybrid priority queue architecture, the hardware priority queue can be extended into off-chip memory and managed in both hardware and software, when the queue size exceeds hardware limits. The priority queue, when managed in hardware, supports constant time enqueue operations and dequeue operations in $O(\log n)$ time. The hardware utilisation of the our priority queue increases logarithmically with the queue size and avoids complex pipelining logic.

One of the limitations of the hardware schedulers described above is that, once deployed, they can only support a fixed number of tasks. Our hybrid scheduler architecture overcomes this limitation by switching between hardware and software modes of operation depending on the number of tasks in the system. The transitions between hardware and software is fast and has low overhead. The hybrid priority queue is used as a part of our real-time scheduler to improve performance and timing predictability.

7 Conclusions and future work

A new hybrid priority queue architecture has been implemented, which can be managed in hardware and/or software. The priority queue when managed in hardware supports enqueue and peek operations in O(1) time, returns the top-priority element in O(1) time, and completes a dequeue operation in $O(\log n)$ time. The design enables quick and low overhead transition between hardware and software management. We utilise hardware logic to enhance the performance of queue operations even when managing the priority queue in software. As an application of the proposed priority queue architecture, a scalable hybrid scheduler is implemented that supports 255 tasks in hardware mode and up to an arbitrarily large number of tasks in hybrid mode. The scheduler when managed in hardware, showed up to 90% reduction in scheduler overhead when compared to the software scheduler. Our results show that the hardware scheduler has 98% less variation in execution time when compared to the software scheduler, thus giving more predictable execution times, which is necessary in high-performance real time systems.

Avenues of future work include,

1 reducing the rate of performance degradation as queue overflows into software,

2 evaluating the use of our hybrid priority queue in discrete event simulation and network optimisation algorithms

3 integrating our hybrid scheduler with Real-time Linux and characterising the scheduler performance.

Acknowledgements

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

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

Bhagwan, R. and Lin, B. (2000) '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*, Vol. 2, pp.538–547.

Bloom, G., Parmer, G., Narahari, B. and Simha, R. (2012) 'Shared hardware data structures for hard real-time systems', in *Proceedings of the tenth ACM International Conference on Embedded Software, EMSOFT '12*, pp.133–142, ACM, New York, NY, USA.

Bumiller, E. and Shanker, T. (2011) 'War evolves with drones, some tiny as bugs' [online] http://www.nytimes.com/2011/06/20/world/20drones.html (accessed February 2014).

Burleson, W., Ko, J., Niehaus, D., Ramamritham, K., Stankovic, J., Wallace, G. and Weems, C. (1999) 'The spring scheduling coprocessor: a scheduling accelerator', *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, Vol. 7, No. 1, pp.38–47.

Chandra, R. and Sinnen, O. (2010) 'Improving application performance with hardware data structures', in *Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW), IEEE International Symposium on*, pp.1–4.

DARPA (2011) 'Nano air vehicle (NAV)' [online] http://www.darpa.mil/NewsEvents/Releases/2011/11/24.aspx (accessed February 2014).

Eberle, H., Gura, N., Shantz, S.C. and Gupta, V. (2008) 'A cryptographic processor for arbitrary elliptic curves over gf (2m)', *International Journal of Embedded Systems*, Vol. 3, No. 4, pp.241–255.

Grossman, L., Brock-Abraham, C., Carbone, N., Dodds, E., Kluger, J., Park, A., Rawlings, N., Suddath, C., Sun, F., Thompson, M., Walsh, B. and Webley, K. (2011) 'The 50 best inventions', *Time Magazine*.

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

Ioannou, A. and Katevenis, M. (2007) 'Pipelined heap (priority queue) management for advanced scheduling in high-speed networks', *Networking, IEEE/ACM Transactions on*, Vol. 15, No. 2, pp.450–461.

ITRS (2009) 'The International Technology Roadmap for Semiconductors (ITRS)', *Lithography* [online] http://www.itrs.net/ (accessed February 2014).

Jones, D.W. (1986) 'An empirical comparison of priority-queue and event-set implementations', *Commun. ACM*, Vo. 29, No. 4, pp.300–311.

Kohout, P., Ganesh, B. and Jacob, B. (2003) 'Hardware support for realtime operating systems', in *Hardware/Software Codesign and System Synthesis, First IEEE/ACM/IFIP International Conference on*, pp.45–51.

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

Liu, C. and Layland, J. (1973) 'Scheduling algorithms for multiprogramming in a hard-real-time environment', *Journal of the ACM (JACM)*, January, Vol. 20, No. 1, pp.46–61.

Moon, S-W., Shin, K. and Rexford, J. (1997) 'Scalable hardware priority queue architectures for high-speed packet switches', in *Real-Time Technology and Applications Symposium, Proceedings, Third IEEE*, pp.203–212.

Mooney, V.J. and Micheli, G.D. (1997) *Hardware/Software Co-design of Run-time Schedulers for Real-time Systems*, Technical report, Stanford, CA, USA.

Morton, A. and Loucks, W.M. (2004) 'A hardware/software kernel for system on chip designs', in *Proceedings of the 2004 ACM Symposium on Applied Computing, SAC '04*, pp.869–875, ACM, New York, NY, USA.

Muller, M., Klockner, J., Gushchina, I., Pacholik, A., Fengler, W. and Amthor, A. (2013) 'Performance evaluation of platform-specific implementations of numerically complex control designs for nano-positioning applications', *International Journal of Embedded Systems*, Vol. 5, No. 1, pp.95–105.

Nakano, T., Utama, A., Itabashi, M., Shiomi, A. and Imai, M. (1995) 'Hardware implementation of a real-time operating system', in *TRON Project International Symposium, Proceedings of the 12th*, pp.34–42.

Ors, B., Batina, L., Preneel, B. and Vandewalle, J. (2008) 'Hardware implementation of an elliptic curve processor over gf (p) with montgomery modular multiplier', *International Journal of Embedded Systems*, Vol. 3, No. 4, pp.229–240.

Park, T.R., Park, J.H. and Kwon, W.H. (2001) 'Reducing OS overhead for real-time industrial controllers with adjustable timer resolution', in *Industrial Electronics, ISIE, IEEE International Symposium on*, Vol. 1, pp.369–374.

Rahmouni, K., Chabanet, S., Lambelin, N. and Petrot, F. (2013) 'Design of a medium voltage protection device using system simulation approaches: a case study', *International Journal of Embedded Systems*, Vol. 5, No. 1, pp.53–66.

Ronngren, R. and Ayani, R. (1997) 'A comparative study of parallel and sequential priority queue algorithms', *ACM Trans. Model. Comput. Simul.*, Vol. 7, No. 2, pp.157–209.

Ronngren, R., Riboe, J. and Ayani, R. (1991) 'Lazy queue: an efficient implementation of the pending-event set', in *Proceedings of the 24th annual symposium on Simulation, ANSS '91*, pp.194–204, IEEE Computer Society Press, Los Alamitos, CA, USA.

Saez, S., Vila, J., Crespo, A. and Garcia, A. (1999) 'A hardware scheduler for complex real-time systems', in *Industrial Electronics, ISIE '99, Proceedings of the IEEE International Symposium on*, Vol. 1, pp.43–48.

Stankovic, J. and Ramamritham, K. (1991) 'The spring kernel: a new paradigm for real-time systems', *Software, IEEE*, Vol. 8, No. 3, pp.62–72.

Varela, M., Cayssials, R., Ferro, E. and Boemo, E. (2012) 'Real-time scheduling coprocessor for NIOS II processor', in *Programmable Logic (SPL), VIII Southern Conference on*, pp.1–6.

Vaucher, J.G. and Duval, P. (1975) 'A comparison of simulation event list algorithms', *Commun. ACM*, Vol. 18, No. 4, pp.223–230.

Zhuang, X. and Pande, S. (2006) 'A scalable priority queue architecture for high speed network processing', in *INFOCOM, 25th IEEE International Conference on Computer Communications. Proceedings*, pp.1–12.