A Hardware Scheduler for FPGA-based Embedded Real-Time Systems

Yi Tang, Neil W. Bergmann

School of ITEE, The University of Queensland, Australia Email: yi.tang@uq.edu.au

Abstract:

A hardware scheduler is developed to improve real-time performance of soft-core processor based computing systems. Since it is implemented on an FPGA, the hardware scheduler overcomes the design inflexibility found in previous accelerators and provides a configurable architecture to support different scheduling algorithms. A new task queue architecture is designed to better support practical task controls and maintain good resource scaling. Based on this, the scheduler supports a time sliced priority scheduler and two deadline based schedulers (Earliest Deadline First and Least Slack Time.) The hardware scheduler reduces scheduling overhead by more than 1000 clock cycles and allows up to 23.7% more application execution time. Scheduling jitter is reduced from hundreds of clock cycles in software to just two or three cycles for most operations. The main disadvantage for the hardware scheduler is the extra resource cost. A typical EDF scheduler developed for small scale embedded system increases circuit area of a soft-core system by 17%.

Keywords: FPGA, task queue, priority scheduling, EDF and LST scheduling

1. Introduction

A real-time system is one where correct operation of the system depends not only on the correct values of system outputs, but also on those outputs being produced in a timely manner. A Real-Time Operating System (RTOS) assists in the operation of real-time software systems by providing basic support for concurrent real-time software tasks, such as scheduling, resource management, synchronization, and inter-process communications [1]. The use of an RTOS is now common in many real-time software applications, and even small-scale embedded system could benefit from using an RTOS [2]. However, an RTOS brings along cost in terms of computational overhead and response-time unpredictability.

Recent advances in Field Programmable Gate Array (FPGA) technology provide the possibility to implement all components of a complex, softcore processor based digital system on a single FPGA chip. This implementation adds flexibility in design and provides a quicker time to market. However, a softcore processor has poorer performance when compared to a hardcore processor. Running an RTOS on such a system can significantly degrade the system's real-time performance.

The flexibility of an FPGA based system provides the possibility to explore new computing architectures to accelerate a system's real-time performance [3]. It also enables designers to fully embrace the FPGA's flexibility to customize the system for individual applications.

Hardware acceleration for operating systems has a long history. Various acceleration techniques, such as caches, DMA and memory management, have now been widely accepted in modern computer systems. The first example of a hardware accelerated RTOS was probably in early 1990s, when the well-known RTU project [4] proposed a hardware kernel covering key RTOS functions. Hardware acceleration techniques for RTOS functions have been proved to reduce computation overhead and improve system predictability [3, 4]. They can also lower the system requirements such as CPU frequency and system memory, and thus reduce cost and power consumption[5].

Scheduling is one of the key factors that affects the behaviour of real-time systems [6]. The scheduler decides which task to run next in order to best meet the system's real-time constraints. Commercial RTOS generally use a simple priority-based scheduler [2] to implement Rate Monotonic (RM) Scheduling since this is easy to build and has a relatively low overhead. Compared to RM, deadline based algorithms (such as Earliest Deadline First, EDF, and Least Slack Time, LST) can potentially provide better decisions. RM requires the designer to include a large safety margin to account for sub-optimal scheduling decisions in worst-case situations. As a result, it can lead to low average system utilisation. Another critical problem is the task starvation problem when a running task misbehaves. On the other hand, deadline-based scheduling has been shown to be optimal in terms of meeting real-time constraints, but incurs a high runtime overhead [7]. Besides that, it also requires the system to establish a system clock and dynamically update a task's scheduling priority at each job release. Further, the scheduler should provide a mechanism to monitor task execution time since explicit deadline information is introduced. These issues mean that a software deadline-based scheduler has a large overhead which reduces its benefits.

A hardware scheduler moves the scheduling function from software into hardware, which runs in parallel with the software tasks. Thus algorithms that require frequent scheduling activity and time awareness can be implemented without preempting running tasks. The disadvantage of a hardware scheduler is increased hardware usage and algorithm inflexibility. However, a small scale embedded system has a limited number of tasks and predefined functions. An FPGA based soft-core system makes it possible to customize the hardware scheduler on an application-by-application basis. This makes a hardware scheduler a feasible scheme for FPGA based applications.

The remainder of the paper is structured as follows. In section 2, the basic theory of real-time scheduling is introduced. In section 3, previous research results in hardware scheduling are reviewed. In section 4, a new hardware scheduling architecture is presented, and in chapter 5 its performance is analysed and compared to software schedulers.

2. Related Works

Many task schedulers are developed based on the task queue model [7, 8], in which tasks waiting for execution are stored in a ready queue. When the processor becomes available, the scheduler fetches the next task from the ready queue and dispatches it to the processor for execution.

When scheduling is implemented in software, RM and EDF are two commonly used algorithms. RM is a static priority algorithm, which means a task maintains its priority throughout the system's lifetime. In contrast, EDF is a dynamic priority algorithm, in which a task's priority changes based on the absolute deadline of the current job. Commercial RTOS generally uses a static priority scheduler to implement RM scheduling. The simpler approach, Bitmap Scheduler [9], implements a bit value priority queue but has the limitation that each priority only holds one task. The multilevel feedback queue model [7] allows each priority level to hold multiple tasks. Its implementation splits the ready queue into several FIFO queues, which increases complexity and runtime overhead if the number of priority levels supported is high.

EDF and LST are two well-known deadline based algorithms. They are both dynamic priority algorithms and have been proved to be optimal, i.e., if a feasible schedule is available, these algorithms will find it [7]. Compared to RM scheduling, dynamic priority schemes are more difficult to implement [7, 10], but they have the advantages that such systems have lower task pre-emption rate and allow better processor utilization [10]. Formal deadline scheduling algorithms require a kernel to provide task level time and deadline management. Currently, most software EDF schedulers are research oriented and do not support effective deadline management for all tasks. Some RTOS implement EDF using Dynamic Priority allocation on top of a priority based scheduler. This scheme, however, can introduce a large worst case computation overhead compared to RM schedulers. Another natural approach implements a ready queue sorted based on task's absolute deadline. This method will be further studied in section 3. LST is even more difficult to implement because it requires the scheduler to continuously calculate the slack time for all tasks. The Laxity-tie problem is another disadvantage which prevents it being used in common RTOS. It defines the situation when two or more tasks have same laxities and task switch happens continuously until the tie breaks.

The advantage of software schedulers is that they are simpler, more flexible and scalable. They also fit a software RTOS naturally, thus designers can implement algorithms in different ways without worrying about the hardware architecture, signal synchronization, resource usage and other complex issues found in hardware design. However, complex algorithms make the scheduling process less predictable and introduce more computation overhead.

Researchers have found that task scheduling can be faster and more predictable if it is separated out of the central processor. The obvious benefits are reduced runtime overhead and event response time due to smaller critical sections [4, 11]. It also saves the system from frequently servicing the system tick Interrupt Service Routine (ISR) [12], which improves scheduling predictability and increases processor utilization. Further, some advanced features like time slicing and deadline management can be easily implemented. The separate scheduling module is achieved either through the co-processor kernel approach or hardware scheduler approach.

The co-processor kernel approach uses either a separate full-feature processor or a similar architecture to run a software kernel program. Early projects include the Spring Co-processor [13], and Kernel Processor [14]. With the advance in multi-processor technology, more implementations can be expected[15]. The main drawback for this approach is the communication and synchronization problem between two individual systems, as stated in [11]. Further, this approach is costly in resources and power, and a dedicated coprocessor interface is required on the CPU. For safety critical applications, the testing and verification process for software is much more complex than for hardware [16].

The hardware scheduler approach uses customized circuits to further improve the real-time performance of the scheduler. Some projects [17] use state machines and common software data structures to implement scheduling algorithms which is similar to the coprocessor kernel approach, and this scheme is referred to as the serial hardware approach. Another category implements dedicated hardware data structures and applies special hardware techniques (like pipeline or parallel processing) to facilitate the scheduling process. These are referred to as the dedicated hardware scheduler approach. Most work on the latter approach is inspired by the success of hardware packet schedulers in network communications [18]. This includes simple priority encoder, comparator tree and various hardware task queue models, such as a FIFO priority queue (PQ), Shift Register PQ and Systolic Array PQ. Because architectures in this approach are specially optimized for scheduling, they always achieve a much shorter scheduling delay than the serial hardware approach. Hardware circuits also enjoy good reliability and runtime predictability. This paper focuses on the dedicated hardware scheduler approach.

The FIFO PQ model, firstly introduced as a hardware scheduler in the RTU projects [19], uses a few FIFO queues plus a priority encoder to implement a hardware ready queue. Another similar approach is found in the Hthreads project [20]. This model is easy to implement and its performance solely depends on the priority encoder. The critical problem, however, is the resource usage. If dynamic priority allocation is allowed in the system, the scheduler's resource can scale quadratically, with the worst case that all tasks occupy a single priority, but each queue is made large enough to hold all tasks. The architecture also has the disadvantage that it doesn't allow tasks to be easily removed from the queue.

The Shift Register (SR) PQ model is used in [12]. The design has a configurable architecture which can perform simple priority, EDF or direct RM scheduling. Each node in the SR queue implements comparison and decision logic to support node removal and en-queue operation. Since a new entry needs to be broadcast to each node, it creates bus loading problems and limits the scalability of the architecture. Also the implementation is greatly affected by the size of scheduling information (SCHD-INFO), N. If N is large, the comparator's time complexity and resource usage will grow significantly. However, many practical systems require both fine time granularity and long system lifetime. These issues lead to a large SCHD-INFO and make the Shift Register PQ model not applicable. Another similar approach, the Systolic Array PQ is implemented in [21]. This circuit removes the parallel input to avoid the bus loading problem. However, this modification increases task enqueue overhead and the resource problem remains. Another important project develops a real time mechanism to circumvent the size and sharing limitation of the hardware task queue[22].

The comparator tree model is one unique approach to implement a hardware scheduler [23, 24]. Different from approaches based on the task queue model, tasks in this model are implemented as parallel Finite State Machines (FSMs). The parallel outputs of task status are linked to a comparator tree to compute the scheduling result. This model costs even more logic resources than the task queue models. The simple priority encoder model is only applicable for the bitmap scheduler[25]. It provides limited functionality and little performance gain when compared to its software counterpart.

Most hardware approaches for task scheduling originate from the packet scheduler. However, packet and task scheduling are two different application scenarios and should have different implementation focuses. Generally, a task scheduler deals with a more complex control model and requires a larger scale on scheduling priority. This issue will be further discussed in section 3.2.

In the different hardware scheduler models discussed above, the SR task queue model provides a balanced solution between processing speed and resource cost. This research modifies the SR task queue model and implements a new task queue model that overcomes the defects. When applied on small scale embedded systems, the number of tasks is limited which relieves the scalability problems of hardware schedulers.

3. Hardware Task Queues

As discussed above, the task queue model can serve as a basic primitive in task scheduling, and is adopted in both software and hardware schedulers. Besides the ready queue in the task scheduler, a task queue is also used to construct other kernel queues [8] like the sleep queue or semaphore queue.

3.1 Task Queue Basics

A task queue data structure is a linked list in which each list node stores a task pointer. Tasks in the queue are linked serially in a particular order. This identifies three types of task queue. In the FIFO and LIFO queue, tasks are arranged by their arrival order. The priority queue, on the other hand, arranges tasks based on the priority (SCHD-INFO). If tasks share equal priority, then they follow the FIFO order.

Two nodes in the queue are of special interest: Q-head and Q-tail. Q-head marks the head of the queue while Q-tail marks the last occupied node in the queue. This data structure includes three essential control operations. The *enqueue* control inserts a new task entry to the correct position of the queue. The *dequeue* control, issued by the queue controller when a task switch is due, shifts out the task in Q-head and Q-head is updated. The *remove* control deletes a task of any position in the queue. Apart from that, kernel can also peek at the data stored in Q-head without removing it.

Many RTOS use the ready task queue to implement task scheduling. In such cases, the scheduler translates input task controls into queue controls, and only the tasks ready for execution are added to the ready queue. The scheduling processes include ready queue management and task pre-emption detection, which compares the priority between queue head and the running task.

3.1.1 Task Queue Implemented in Software Scheduler

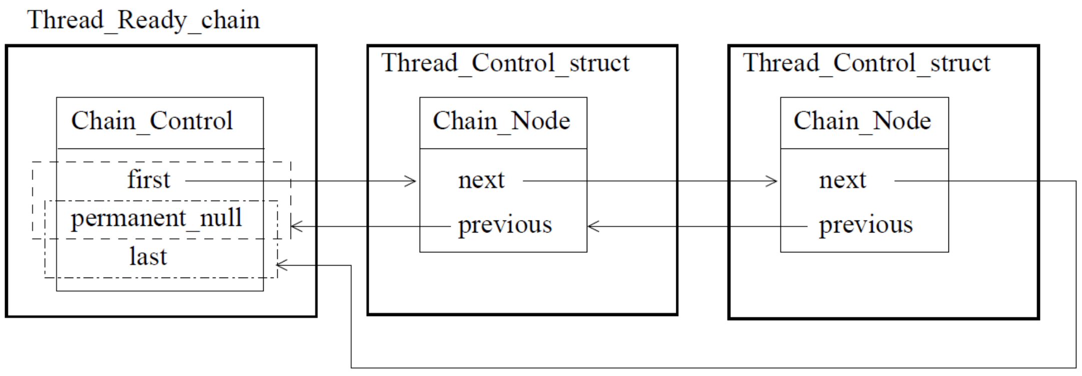

Figure 3-1 The RTEMS Ready Queue [26]

Figure 3-1 shows a FIFO ready thread queue implemented in RTEMS [26]. A thread is considered the same as a task in this paper. In this software model, queue nodes are linked together through software pointers. Each queue node represents a particular thread, except the first node which is a control field of the thread queue. The first node contains two pointers that map the Q-head and Q-tail. With these two pointers, a scheduler can easily locate the Q-head to perform *dequeue* and Q-tail to *enqueue* a new task. Other nodes also maintain pointers that connect the adjacent nodes. In each thread structure, it has one field that records the node address of the assigned kernel queue. Normally a thread can only exist in one kernel queue, thus this field will be updated when its state is changed. With that, the scheduler can easily locate the thread node and perform the *remove* control. During node removal, the scheduler only needs to reset the pointers in two adjacent nodes.

3.1.2 Task Queue Implemented in Hardware Scheduler

The Shift Register PQ has been implemented as the ready task queue in [12], as shown in Figure 3-2. It is an array of shift register blocks that is self-sorted based on the priority of queue blocks. Besides the priority, a queue block also stores an address field, which is the task identifier (TID) if implemented as a task queue. Nodes in the queue have two parallel entries. The new_entry_bus is used to enter queue data while another input issues control commands (read and write). Besides that, each block is connected with other blocks next to it and the stored data could be shifted left or right by command. Tasks in the queue are sorted based on their priority and the 0th block holds the current highest-priority task.

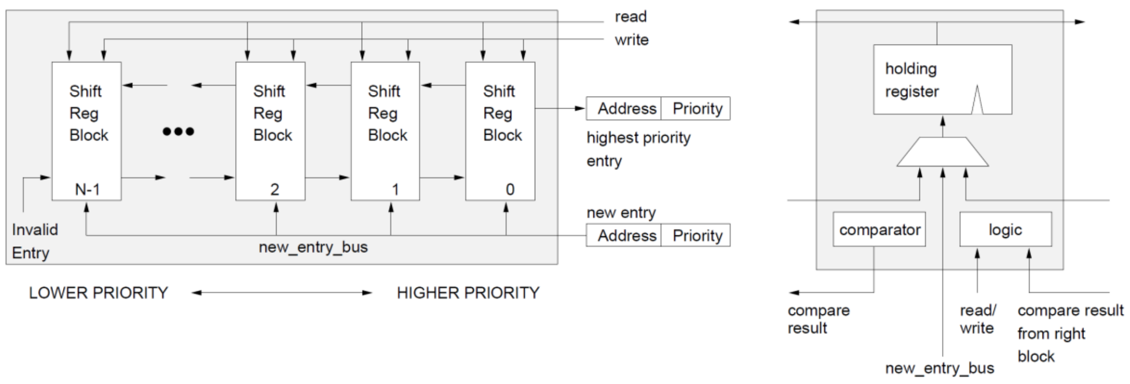

Figure 3-2 Shift Register Priority Queue and Shift Register Block[18]

This model only supports the *dequeue* (read) and *enqueue* (write) controls. Based on the controls, the queue block has three operational modes: keep current task, shift left/right and latch the new task. On an *enqueue* operation, a new task and its priority are broadcast to all the blocks via the parallel entry. Priority of the new task is compared with that of the local task through the comparator and each block makes a local decision. In the end, only one block will latch the new task. The other blocks will either stay still or do a shift operation to latch their right neighbour's data. This leads to a net effect that forces the queue to automatically re-order itself. When a *dequeue* control is issued, the data in 0th block is read and all other blocks are shifted to their right.

This approach utilizes distributed control to process queue controls, which reduces the overhead of the *enqueue* operation. Both priority and deadline based scheduler can be constructed with this circuit. The bus loading problem is not critical if the scheduler is applied to small scale embedded systems. However, some other defects stop it from being widely accepted, as will be discussed in section 3.2. This paper modifies this circuit and presents a new approach that utilizes centralized control to process *enqueue* control and distributed control to manage other queue controls.

3.1.3 Scheduling based on Ready Task Queue

The ready queue model can be applied to many scheduling algorithms. Its main idea is to let all ready tasks be prioritized in a queue in advance. When task switching is due, the scheduler only needs to output the queue head. Its core function is the *enqueue* operation which inserts a task into a proper position in the queue. For FIFO and LIFO queues, the *enqueue* process is straight forward, and the scheduler simply updates the pointers that reference Q-head and Q-tail nodes. For a priority queue, the *enqueue* operation is more complicated and different approaches can be used to accelerate this process.

The architecture of traditional hardware queues makes it difficult to implement the *remove* control. A software queue implements a virtual queue, in which task nodes are connected through pointers. The scheduler can easily identify a task node in the queue, and process a *remove* control. In contrast, a hardware queue implements a queue in real circuits. Task shifts caused by queue controls are not visible to the controller. This makes it hard to identify the actual position of a task in the queue. Thus the *remove* control is not feasible in such an architecture.

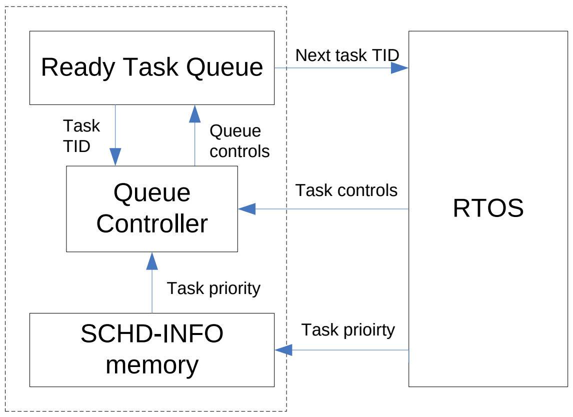

Figure 3-3 Hardware Scheduler built with Task Queue

Figure 3-3 shows a simple functional model of a priority based hardware scheduler. The Ready Task Queue implements a priority queue that is sorted based on task priority. Thus the scheduler generates a result by simply fetching the head of the Ready Task Queue. The Queue controller is responsible for translating task controls into queue controls and processing each queue control. When a task is activated, its priority is stored in the SCHD-INFO memory and the task is inserted into the ready queue. In this model, the RTOS fully controls the scheduling process. When a task control is raised, it retrieves the Q-head information and decides whether task pre-emption is needed.

3.2 Task Queue vs. Packet Queue

The Shift Register PQ model discussed in [18] was originally developed for packet scheduling in computer networking. Its design prerequisites are quite different from that of task management in an embedded RTOS. In packet scheduling, the control operations for the hardware queue can simply be data insert (or *enqueue)* and *dequeue*. In contrast, the *remove* control is necessary for a task queue.

In a practical task scheduler, the *remove* control contributes to various functions. Firstly, a system monitor task or an ISR can use it to block a ready task that has malfunctioned. Meanwhile, some RTOS features require tasks' SCHD-INFO to be dynamic, for example the dynamic priority allocation in EDF and priority inheritance scheme. The *remove* control can facilitate the modification of SCHD-INFO online. Moreover, *remove* control is useful when a task queue is implemented in other formats, such as a sleep queue or semaphore queue.

Besides the queue control issue, there are other design considerations that vary between these two applications.

1. A complex algorithm may require the scheduler to constantly access tasks' SCHD-INFO while the packet queue is not accessible except through dequeue control.
2. A task queue could have a small queue length (less than 32 in small scale embedded systems) while a network packet queue normally requires a large queue length.
3. Network queues need to consider peak arrival rates while task controls normally arrive at a fairly low rate.
4. A task's SCHD-INFO may have a wide index range (deadline scheduling), while a network queue only uses a small number of priority bits.

Another important difference is resource constraints. In network communication, scheduling could be performed on dedicated hardware without considering its cost and power usage. In an embedded SOC system, however, designers need to restrict the circuit resource use, which then leads to lower cost and power usage. In an FPGA-based platform, one additional requirement is to better exploit on-chip resources. Thus some memory primitives normally implemented using registers shall be transferred to BRAM [27].

3.3 Basic SR Task Queue

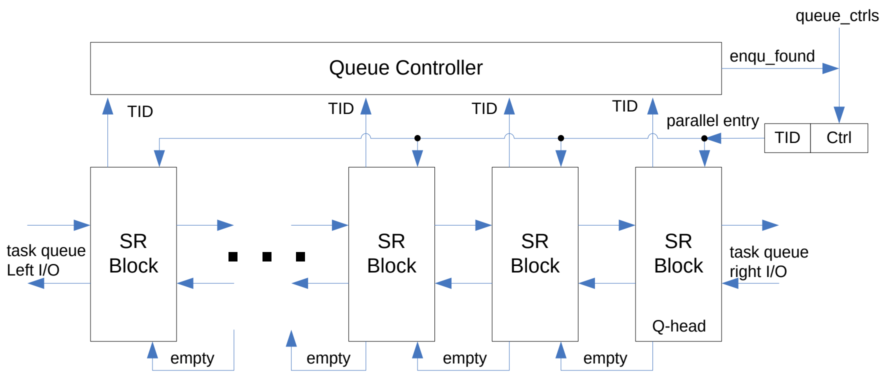

Figure 3-4 Basic SR Task Queue

The Basic SR (BSR) task queue model has been developed based on the Shift Register PQ model. It adds a remove control clause and is resource sensitive. As shown in Figure 3-4, the model has a similar task block array architecture with serial and parallel entries. However, each node contains an extra parallel TID (only used in priority queue) and an empty flag propagation line. Similarly to the

Shift Register PQ, each block in the BSR queue has its own logic cell and performs distributed control based on parallel entry and its stored TID. The Q-head in this model represents one physical end of the queue, and the dequeue control outputs the Q-head whose TID becomes the new running task in the system. This primitive architecture natively records tasks based on their arrival order and completes all task controls within two clock cycles. It can be used to implement FIFO, LIFO and priority queues in a hardware scheduler.

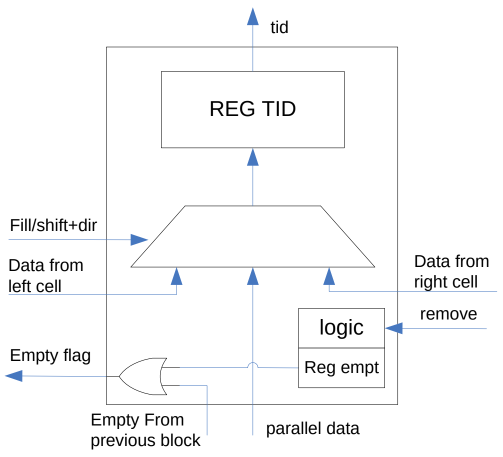

Figure 3-5 BSR Task Queue Block

Unlike its predecessor, a task block in the BSR queue only contains some simple logic and registers for the TID field and an empty flag. The queue controller performs centralized enqueue control, thus the SCHD-INFO and relative comparator are no longer implemented in the task block. The empty register, which marks a node empty, is connected to a propagation chain and generates the empty flag. They are added to facilitate *remove* control and mark the Q-tail node. Generally, a node identifies itself as a Q-tail when it is empty but receives a non-empty flag from its right block.

The new control clause, remove control, is processed in two cycles. In the first cycle, each task block will compare its REG-TID field with the parallel TID input. If the target task is found, the block will set its empty register which also causes its empty flag raised and propagates the left end of the queue. In this case, if one block in the middle of filled blocks is removed, all its left blocks will output the empty flags. In the second cycle, the queue automatically reorders itself. All SR blocks with empty flag raised will copy their left block at the same time. This operation then fills the deleted block in the queue.

FIFO based Task Queue

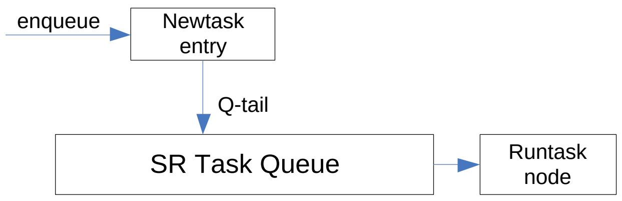

Figure 3-6 FIFO Task Queue

For the FIFO queue, a task is inserted to the queue through the parallel entry. During the process, the *enqueue* control is asserted and new task TID broadcast to all queue blocks. The current Q-tail block will record the new task TID. It also clears the empty flag thus its right neighbour block becomes the new Q-tail block. With the *dequeue* control, the queue head will be fetched and the whole queue will be shifted left. Similarly, a LIFO queue is also easily implemented with BSR queue model, except that its enqueue/dequeue controls and data are addressed through Q-head.

Priority Task Queue

The BSR Task Queue could also be used to construct a priority task queue like the one in the example scheduler architecture in Figure 3-4. The priority task queue is sorted based on a task's SCHD-INFO, which could be either priority or deadline. Unlike the Shift Register PQ model, the SCHD-INFO in BSR queue is stored externally in a dedicated memory and can be accessed by the queue controller using the index TID. Moreover, enqueue sorting is processed using centralized computing through the queue controller.

During the *enqueue* operation, the queue controller needs to search the queue to find the right position to insert the new task. In the queue search process, it goes through the queue with a certain search algorithm. The queue controller reads the TID field in the selected queue node and uses it to fetch the SCHD-INFO, which is compared with the new task's SCHD-INFO. This search can use either a linear or binary search[28], as discussed in section 5.3.

When queue search is performed , the queue controller reads the TID field of all occupied task blocks through a TID bus. Once the enqueue destination is found, a target node is selected to latch the new task. Thus an *enqu_found* signal is broadcast along with target node's current TID. This lets the queue to reorder and prepare an empty node for the new task. In the next cycle of *enqu_found*, the new task TID is stored.

With the centralized *enqueue* control, the SCHD-INFO comparator logic is moved to the queue controller. Because the SCHD-INFO is stored in a dedicated memory block, it is possible for a task to have SCHD-INFO with a large index range such as 32bits. This fits the requirements of the deadline scheduling. Otherwise, having SCHD-INFO register and comparator logics in each queue block will lead to high resource cost.

3.4 Advanced SR Task Queue

Priority Selector & Priority Multiplexer

A hardware circuit can be used to filter a number of inputs and output the active one with highest priority. A priority selector (PSEL) has N data inputs and N data outputs. Its input lines follow a priority order and the input having the highest priority will take precedence, while other inputs get filtered. Table 3-1 shows an example of a 4 bit priority selector.

Table 3-1 4-bits Priority Selector

I3	I2	I1	I0	O3	O2	O1	O0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	x	0	0	1	0
0	1	x	x	0	1	0	0
1	x	x	x	1	0	0	0

The logic equation for the priority selector is listed below. The N bit priority selector has inputs I_i and outputs O_i. C_i is an intermediate signal that propagates from the input with highest priority. The output of the device depends both on the input and the intermediate signal.

$$C_{i+1} = C_i + \overline{I_i} \; (i = 0, 1, \ldots N\text{-}1) \tag{6.1}$$

$$O_i = I_i + C_i \; (i = 0, 1, \ldots N\text{-}1) \tag{6.2}$$

The priority selector is a primitive circuit which can be used to build a priority encoder and priority multiplexer (PMUX). There are various approaches to implement the priority selector, either optimized for speed or for logic resource use. As the current project is prototyped on an FPGA, the intermediate signal is implemented based on propagation chain (or carry chain [27]) of the device.

A PMUX is constructed by combining a multiplexer with a priority selector. As illustrated in Figure 3-7, it has N data ports and each data input comes with a separate select line. Similar to a priority encoder, the select lines imply a priority order. The PMUX outputs the data port enabled with the highest priority.

Advanced SR Task Queue

The Advanced SR (ASR) task queue is based on the BSR task queue and is optimized specifically for priority queues in deadline scheduling. As we can see in the BSR model, the most time-costly operation in the sorted task queue is the queue search process, incurred by the *enqueue* control. The Advanced SR task queue introduces two new queue control clauses and incorporates not only ready tasks but also blocked tasks. With this approach, synchronization related task controls (task activation and blocking) will not cause tasks to enter and leave the queue frequently. Thus reduces the occurrence of the queue search process.

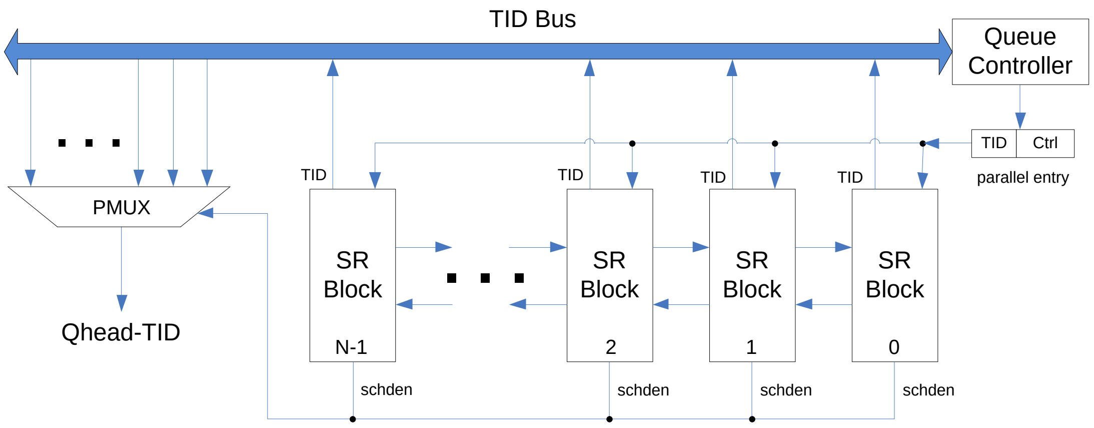

Figure 3-7 Advanced SR Task Queue Model

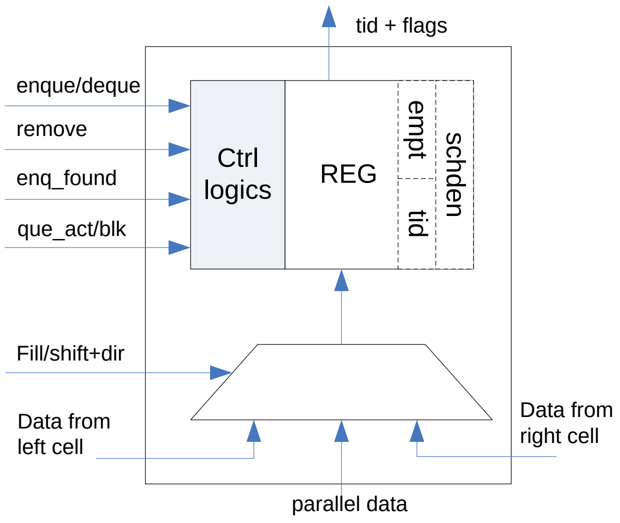

Figure 3-8 Advanced SR Task Queue Block

To distinguish between ready tasks and blocked tasks, a *schden* flag is introduced. Two extra queue control clauses, *que_act* and *que_blk*, are applied to control the *schden* flag. Similar to the *remove* control, each SR block checks the parallel entry when activate or block control is issued. If the TID matches, its *schden* flag will be set or cleared. These two control clauses take only one cycle for the scheduler to process. This scheme reduces the occurrence of enqueue operations when compared to BSR task queue.

Other control clauses are similar to the priority queue in the BSR model. But the Q-head in this model is different from the one in BSR model. Here Q-head is assigned to the “ready” task block with smallest SCHD-INFO (highest priority), thus its physical position is volatile. The queue blocks' *schden* flags are connected to a PMUX, which decides the Q-head. This ensures only ready task is output for task switching.

4. Hardware Scheduler Architecture

This section introduces a configurable scheduler architecture that supports both priority and deadline scheduling algorithms. It also describes its interface with the RTOS.

4.1 Passive vs. Active Scheduler

When cooperating with a software kernel, the hardware scheduler can behave either actively or passively. For a passive scheduler, it does not inform the RTOS that a task switching is due. The scheduling process in the system is fully controlled by software kernel. It issues scheduling control or read scheduler register only when it considers switching might happen. In contrast, an active scheduler decides task switching in the system. It notifies the RTOS, usually through ane interrupt, when a switching is due. The RTOS will quickly respond to the interrupt and performs task switching provided it is not in a critical section.

Many hardware schedulers found in the literature are passive schedulers. The hardware scheduler normally operates with a very low runtime overhead. In a traditional computing system, all task controls are issued by the software kernel, thus RTOS knows that a possible task switching will happen. It then stalls the processor for a few clock cycles and controls the passive scheduler to output a result. The system response time depends on how often the RTOS queries the scheduler.

The active scheduler is normally used as part of a hardware kernel. Many hardware kernels [20] include task management mechanisms, such as task sleep control and interrupt activated tasks. Thus task controls can also be sourced from hardware circuits. In such cases, a software RTOS has limited functionality and cannot foresee every switching point. The hardware kernel instead, decides task switching and an active scheduler becomes essential. The scheduler architectures implemented in this paper are mainly designed to support the active mode, but can also be modified to support passive mode.

4.2 Scheduler Architecture

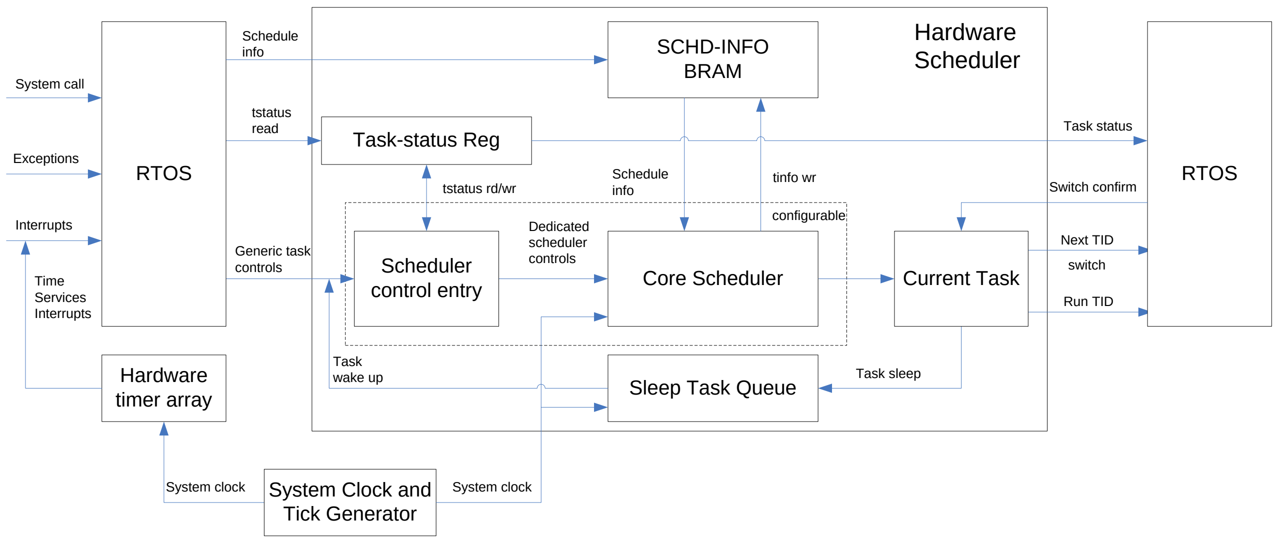

Figure 4-1 Hardware Scheduler Architecture

Figure 4-1 illustrates a simplified hardware kernel that mainly addresses the scheduling operation. This architecture replaces the system tick ISR and software time services with the hardware timer array and Sleep Task Queue (which can be easily constructed with the BSR task queue). Without these two modules, the pure hardware scheduler can also be interfaced with a normal software RTOS. However, removing the system tick ISR can help to maximize the performance gain of using a hardware scheduler. Some other hardware kernel circuits can further enhance system performance when combined with a hardware scheduler, but they are not the focus of this paper.

In the above model, the software RTOS still implements a Task Control Block (TCB) and governs task management. The hardware scheduler maintains task status and performs scheduling based on task controls from the RTOS. The scheduler requires a direct input of system clock, because some scheduling algorithms, such as EDF and LST, need a system clock to calculate the SCHD-INFO for different tasks.

The hardware scheduler also includes memory modules storing SCHD-INFO and task states, the Current Task node exporting the scheduling result, the Scheduler Control Entry that translates task controls and the Core Scheduler that performs core scheduling functions. Because our application is based on a Xilinx FPGA, the on-chip block memory BRAM [27] is used to implement the SCHD-

INFO memory. Each BRAM has two ports, thus one is used by the RTOS to configure SCHD-INFO, and the other provides data for the Core scheduler.

The section enclosed by the dashed line in Figure 4-1 is the configurable region. The actual circuits there depend on the scheduling algorithm implemented. The Scheduler Control Entry module is responsible for checking and translating generic task controls from the RTOS into dedicated controls for specific scheduler architectures, such as block priority level, or dequeue. It also updates internal task states based on control commands. The most important module, the Core Scheduler, implements dedicated hardware scheduler circuits based on the task queue model. It computes the next-to-run-task from initiated tasks and outputs it to the Current Task node.

4.3 Scheduler Interface

In this paper, the hardware scheduler is linked on the processor on-chip bus to minimize communication delay. With the Leon3 processor selected, the scheduler module is connected on the AMBA-AHB bus. The scheduling result can be passed to RTOS either actively or passively. If active scheduler model applied, the switch signal will drive an interrupt that triggers task dispatching. Otherwise, it sets a switching due flag sampled by RTOS after a task control system call. The *switch_confirm* signal is used to synchronize between scheduler and RTOS, since RTOS response can be delayed when system or an application enters a critical section.

The above scheduler architecture has a clear control interface. Since programmed in VHDL, it can be easily configured to support a certain scheduling algorithm and behave either actively or passively. Other complex algorithm can be supported so long as the algorithm can be mapped onto a task queue model.

5. Core Scheduler Implementation

This section discusses the hardware implementation of several scheduling algorithms. It starts with fixed priority scheduling for RM algorithm, in which each task’s SCHD-INFO is constant through its lifetime. Two varieties of priority scheduler are implemented. They are the standard algorithms used in software RTOS. Then two deadline based schedulers are presented. They are more complex algorithms since SCHD-INFO becomes variable. Tasks’ SCHD-INFO in EDF changes every task job while that of LST changes every system tick. These require a more complex architecture for the scheduler.

5.1 Priority Scheduling

Priority-based scheduling is widely adopted in real-time systems [2]. Designers use the RM algorithm to decide a task's priority and this will stay constant during the task's lifetime. Some deadline scheduling algorithms are also mapped to the priority-based scheduler [10], which uses a mechanism called dynamic priority allocation. The priority scheduler discussed in this section cannot independently process such algorithms. Nevertheless, the schedulers are constructed based on the BSR queue, thus supports online change of task priority. It only requires the kernel to implement a top level software scheduler that calculates task priorities at runtime.

5.1.1 Multi-level Feedback Queue Scheduling

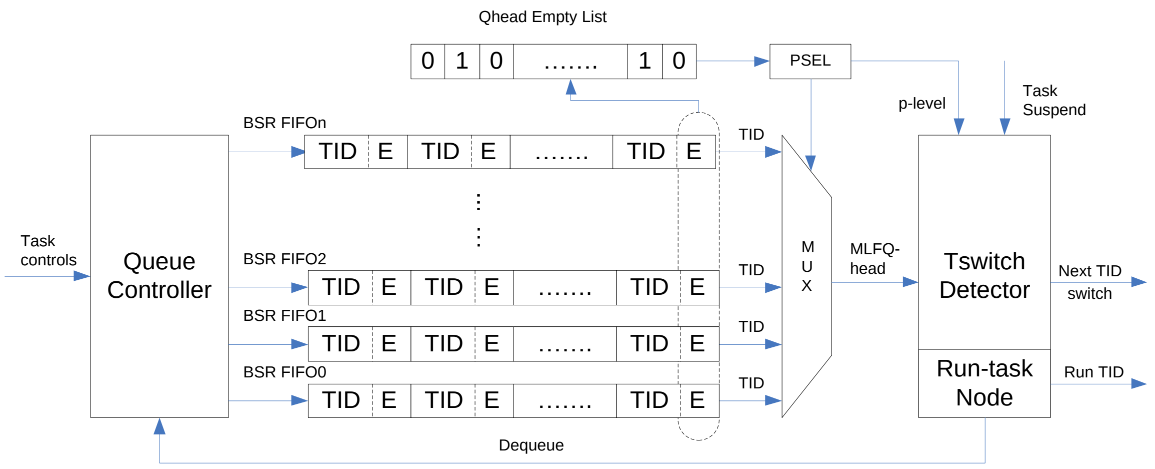

Figure 5-1 MLFQ Scheduler

Some systems allow many tasks to share the same priority level. Such a task set can be scheduled with a Multi-Level Feedback Queue (MLFQ) architecture. This architecture only stores ready tasks and scheduling outputs the task with highest priority. If a number of tasks share the same priority, their schedule follows FIFO order. Figure 5-1 shows a diagram of the MLFQ scheduler. Each priority level is built up with a BSR FIFO Queue, and the queue head is positioned at the right end of the queue. The Queue Controller dispatches new task and queue controls according to the priority information (PID) stored in task control commands or SCHD-INFO BRAM (refer to Figure 3-4). Once a task is activated, an *enqueue* control is dispatched to the task queue with given PID. If it is blocked, the queue controller broadcasts a *remove* control with its TID, and the BSR queue removes the task automatically. When the Tswitch Detector identifies task switching, it sends back a *dequeue* signal (with PID) to shift out the Q-head in the selected task queue.

The Q-head of each task queue outputs its empty flag to the Queue Empty List, which then passes a PSEL to select a non-empty queue with the highest priority level. The Tswitch Detector maintains one register to store the current PSEL output priority level, and a Run-task node that stores information for run-task (PID and TID). When task activation happens, the Tswitch Detector compares two PID values to monitor task pre-emption. Task switching may also occur when the Run-task node receives a task suspend control. If switching is detected, a *dequeue* control is issued to the selected task queue. The Q-head TID then replaces TID in Run-task node and outputs its scheduling results. Meanwhile, the old run-task TID is fed back to the task queue of its priority level (if not blocked). This operation is also known as queue rotate.

5.1.2 Time Sliced MLFQ

Time-slicing allocates equal execution time to eligible tasks of the highest priority. It is only meaningful for algorithms that allow tasks sharing the same scheduling priority. Popular scheduling algorithms that support time slicing are Round-Robin Scheduling and time-sliced MLFQ. For the time-sliced MLFQ, ready tasks on the same priority would be executed in turn.

Normally, a low frequency system tick is routed into the hardware scheduler to provide a time-slice switch signal. Besides task suspend and task pre-emption, this signal can also trigger a *dequeue* operation. In some systems, one time slice can be configured for a number of system ticks, and this number is called the time slice quota. When pre-emption happens in the middle of a system tick, scheduler designer can apply the scheme either to let new task inherit the old task's quota or process a background task.

Time slicing generally introduces more task switching to the system, but this problem can be relieved with a hardware acceleration technique called task pre-loading. In time slicing, the time point of most task switching is pre-known to the system, and so the system can pre-load the new task's context to reduce task switching overhead if such a mechanism is available in the processor. The scheduler can use a P-load node (similar to run-task node) to provide the preload TID. Unlike the run-task node which gets refreshed only when switching is due, the P-load node will be consistent with the Q-head of the highest priority queue.

5.2 Deadline Scheduling

In a fixed-priority scheduler, a task holds constant SCHD-INFO (priority) at all times. However for deadline scheduling, tasks may have different SCHD-INFO at different times. Thus tasks frequently enter and leave the ready task queue. Additionally, their SCHD-INFO may require a large index range.

In the EDF algorithm, a task's SCHD-INFO (absolute deadline) is calculated based on its job release time, which changes every task job. The simple definition of task job is a single round of task execution.

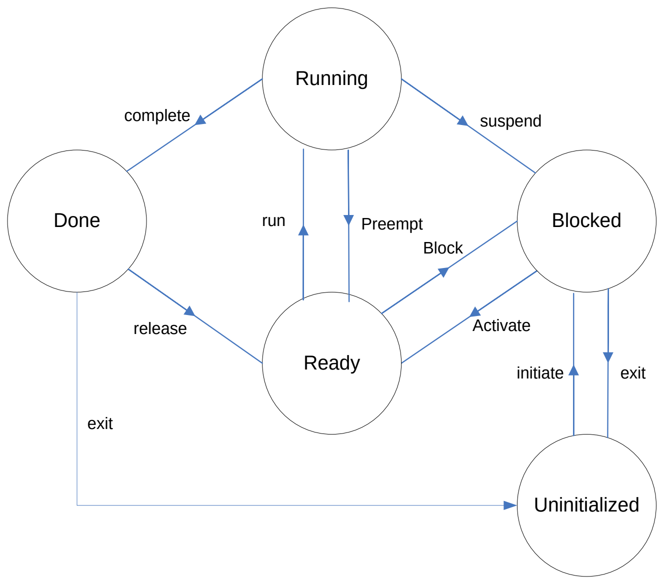

Figure 5-2 Extended Task State Transition Model for Deadline Scheduling

To describe this process, an extended task state transition model is introduced. Different from common task state model, two extra control clauses, "**release**" and "**complete**", are used to mark that a task job is released or completed. The added task state "Done" represents the state that a task job has finished and waiting for another to be released. In EDF scheduling, once a new job is released, the scheduler will calculate its SCHD-INFO based on the relative deadline (pre-known) and current system time. LST Scheduling is more complex, as tasks' SCHD-INFO (slack time or laxity) may vary during execution. This will be discussed later in section 5.2.2.

5.2.1 EDF Scheduler

A scheduler with the EDF algorithm dispatches the task with the earliest deadline. One design option is to construct a ready task queue with the BSR queue that sorts task based on its absolute deadline. The scheduler picks the next executable task from the queue head. With this model, the main overhead comes from the *enqueue* control. In the worst case, inserting a task into a sorted task queue requires *N* steps, where *N* is the length of task queue.

There are two ways to accelerate this process. One is to use a search algorithm to optimize the queue search process, as to be discussed in section 5.3. The other way is to reduce the situations that incur an *enqueue* related queue sorting.

For tasks in the above scheduler model, the *enqueue* process happens in two situations. One comes with the job release of a task, and is called release enqueue. The other one, referred as synchronization enqueue, occurs when a task reactivates from the blocked state. Release enqueue updates a task's SCHD-INFO and inevitably leads to a queue sorting. The synchronization enqueue, however, doesn't change a task's SCHD-INFO. Its relative scheduling priority is maintained until the current job completes. In that case, queue resorting can be prevented.

The proposed EDF scheduler is constructed with the ASR task queue that holds all tasks released, either blocked or ready to execute. Thus a synchronization related task controls will not cause the queue to re-order itself.

To further accelerate scheduling, tasks are differentiated between periodic and aperodic tasks. For all tasks, they have a consistent relative deadline (*rlv_ddl*) since initialization. The other factor to calculate absolute deadline, job release time, is acquired differently. For a periodic task, its release time is pre-known, thus scheduler can update its absolute deadline (*abs_ddl*) before a new job is released. For an aperiodic task, its release time is unknown, thus its *abs_ddl* cannot be decided until a new job arrives.

In such cases, different control commands are applied to task controls (complete and release) used in periodic and aperiodic tasks. This can optimize the scheduling overhead, as will be shown in the results section. For a periodic task, it gets dequeued from the task queue once it completes its current job. The scheduler then calculates its next *abs_ddl* and immediately adds it back to the task queue. With this method, if the task's new job holds a shortest *abs_ddl*, the release operation simply set the *schden* flag in ASR queue, and the switching overhead could be reduced to a minimum.

In contrast, an aperiodic task remains outside the task queue from the time it completes the current job until the next job is released. The release enqueue covers the overhead for the scheduler to calculate the new *abs_ddl* and insert it into the right position of the task queue. If the aperiodic task holds a shortest *abs_ddl*, the switching overhead will be large.

In the proposed EDF scheduler, task init, job release and complete controls for periodic and aperiodic tasks are treated differently and enqueue occurs at different times. The init, complete and release control for periodic tasks are named as *p_init*, *p_cmpt* and *p_rels* while the ones for aperiodic tasks are named as *ap_init*, *ap_cmpt* and *ap_rels*. For a periodic task, enqueue happens immediately after the *p_cmpt* control and *p_init*. The *p_rels* is regarded the same as the task activate control. For an aperiodic task, enqueue happens only after *ap_rels* is issued. The *ap_cmpt* control removes the executing task from the Core Scheduler and *ap_init* performs no action.

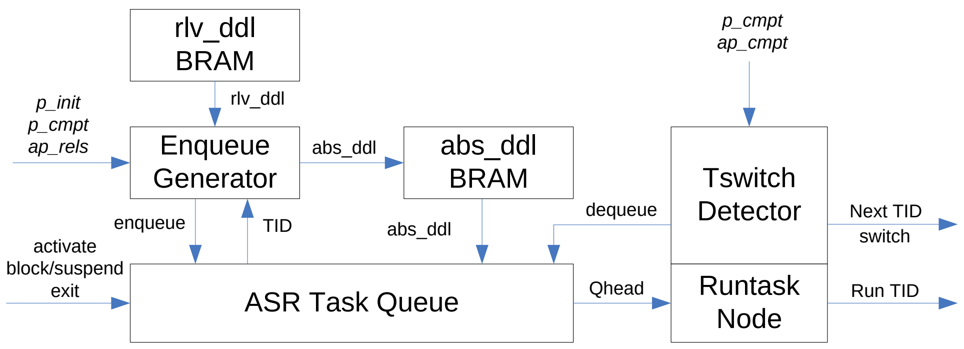

Figure 5-3 EDF Scheduler with ASR Task Queue

Figure 5-3 illustrates the architecture of the EDF scheduler. The enqueue generator processes enqueue-related task controls and issues enqueue controls to the queue controller in the ASR queue. It also calculates the *abs_ddl* and stores them in the abs_ddl BRAM. In this model, two memory blocks are used. The *rlv_ddl* of a task is configured by the software kernel during task initialization and stays constant. In contrast, the *abs_ddl* is used as SCHD-INFO and is constantly accessed by the ASR queue controller.

Similar to the MLFQ scheduler, the Tswitch Detector is used to identify task switching. But in this scheduler, the running task is not shifted out of the queue during execution. Instead, Q-head directly points to the task due to run and Run-task node copies the Q-head info. Dequeue control is issued when a task completes its current job and its *abs_ddl* becomes invalid.

5.2.2 LST Scheduler

Software implementation of the LST algorithm is more complex than EDF. In EDF, task deadlines stay constant once assigned. Therefore, EDF does not require the RTOS to perform scheduling until a new task job arrives. In LST, however, every task's slack time changes with time and pre-emption can occur anytime. In common design, a system tick ISR is required for the software scheduler to update all ready tasks' slack time and perform scheduling at each system tick. This leads to a huge scheduling overhead. However, the LST algorithm can be simplified to ease the implementation.

$$t_{slack} = (d_a - t) - c' = (d_a - c') - \ t = tmval - gclock \tag{6.3}$$

In the equation above, gclock (or t) represents the system's real-time clock. It increases with every system tick. There are two fields in *tmval*. The task's absolute deadline (d_a) stays constant once the job is released. c' is the remaining computation time for a task. When a task gets initiated, c' is assigned with c_{max}, which refers to the worst-case task execution time. The c' becomes variable only when the task is executing and it decreases every system tick then. For a task set, only the running task increases its *tmval* value. All other tasks' *tmval* remain static as time progress. As *gclock* is same for all tasks, the *tmval* can be used as SCHD-INFO instead of slack time. This simplifies the scheduling process, in which scheduler only needs to update the executing task on every system tick.

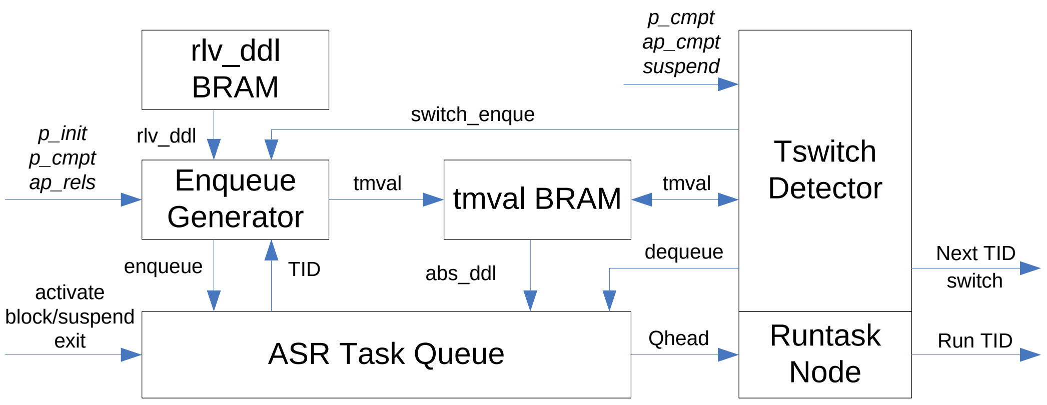

Figure 5-4 LST Scheduler constructed with ASR Task Queue

Similar to the EDF scheduler, the ASR task queue is used to implement circuits for the LST algorithm. One major difference is that the running task is not in the ASR queue, but is placed in the Run-task node, and a counter is attached to it. Therefore, the SCHD-INFO of the running task gets increased every system tick. For non-running tasks, their relative priorities (or SCHD-INFO) stay constant after job start, thus their positions in the ASR queue remain constant. As normal, the Q-head holds executable task with the smallest *tmval*. During scheduling, the Tswitch Detector compares the Run-task node's *tmval* with that of Q-head to detect task pre-emption. Both task pre-emption and suspension can cause a task switch.

Task pre-emption happens in the following process. Initially, the Run-task node holds the smallest *tmval* in the task set. As time progress, its *tmval* gets increased until it is greater than the one in Q-head. Then task pre-emption occurs, it initiates a dequeue control followed by an enqueue control. The original Q-head is shifted to the Run-task node and the running task is added back to the task queue. In such a situation, a laxity-tie happens as these two tasks hold the same or similar slack time. Then these two tasks will run in turn till one of them completes its current job.

There are two common approaches to alleviate the laxity-tie problem. The simple one is to introduce a laxity-tie quota Q_{lt}. When scheduler compares the run-task with Q-head, pre-emption will not happen unless *tmval* (Run-task) > *tmval* + Q_{lt} (Q-head). If a laxity-tie happens, the tied tasks will get executed in turn with a time slice Q_{lt}. Another approach is to implement the Enhanced LST algorithm [23], which can be one direction of future research.

5.3 Enqueue Search for sorted Task Queue

As discussed in the previous sections, the enqueue control for the task queue inserts a task into a sorted priority queue. Because this process is managed by the queue controller with the centralized control model, complex search algorithms can be applied. In this paper, only linear search and binary search are examined.

Linear Search

Linear search or serial search is the easiest way to go through a sorted queue. In this approach, the queue controller polls every block in the task queue. For a task queue with length N, the worst case process time for the controller to walk through the queue is N. Considering SCHD-INFO stored in BRAM, whose access time is two cycle, the worst case process time is $N+2$ system cycles.

Binary Search

Binary search can greatly improve the worst case process time of the enqueue operation. At each stage of the algorithm, the queue of current stage will be divided into two subset queues that are half in size. The controller then compares enqueue task's SCHD-INFO with that of the middle block of the queue. Based on the result, it then decides which subset queue will be used as the queue for next stage. Such operations are repeated till a position found for the enqueue task.

The worst case process time for this algorithm is ceiling($\log_2(N)$) steps. However, we need to consider that the BRAM access time is two cycles. Thus the time complexity is increased to $O(2.\text{ceiling}(\log_2(N)))$. To minimize the processing time, the queue architecture and algorithm needs to be slightly modified to adopt a pipelined architecture.

The BRAM storing SCHD-INFO should be duplicated to create a three port memory block. Thus the controller can access two blocks' SCHD-INFO at a time. In the original algorithm, the subset for next stage will only be judged after the middle block's SCHD-INFO is fetched and compared. Then the controller knows the middle block for the next stage. This requires two cycles to process a stage. In the new algorithm, next stage's middle block is pre-fetched before a subset queue decision is made. And the middle block for next stage has only two possible positions. Through that, two possibilities for the middle block's SCHD-INFO are known at the start of a new stage. And the true value is already computed in previous stage. With that pipeline architecture, the worst case process time for binary search can be reduced to ceiling($\log_2(N)$) +1.

5.4 Deadline Miss Management

Unlike software approaches, deadline miss management in the proposed hardware scheduler is fairly simple. For the EDF scheduler, all tasks assigned absolute deadlines are placed within the ASR task queue. Thus we only need to check the queue node with smallest absolute deadline, which is always the physical head of the task queue. Unlike Q-head, the physical queue node could hold either ready or blocked tasks. This makes sure that scheduler is aware of the deadlines for all released jobs in tasks.

For the LST scheduler, the slack time can be used to foresee whether a task will finish within its deadline, and system can apply special mechanism to prevent deadline miss or transient overload[10]. Since ASR queue uses *tmval* as its SCHD-INFO, the actual slack time needs to be calculated together with the system clock.

6. Performance Evaluation

This section discusses the speed performance of the implemented hardware schedulers. It includes control overhead of operations in the hardware scheduler and the operating frequency.

The computing system is based on the Leon3 soft core processor [29], and is implemented on Xilinx Virtex-5 series FPGA (XUPV5-LX110T). To maximize system predictability, the Leon3 processor is configured to have no data and instruction cache, and 8 register windows. Both data and instructions are stored in on-chip memory blocks (access time 2 clock cycles). Without a hardware scheduler, the system can operate at a maximum frequency of 129MHz.

Scheduling Overhead and Jitter

Table 6-1 illustrates the control process overhead in different schedulers. To compare the performance, a custom software EDF scheduler was developed based on the eCos kernel, which also follows the sorted ready queue model and task control model in Figure 5-2. Together with the eCos MLFQ scheduler, the software scheduler based systems are tested with an on-chip hardware timer to measure task control overheads. In the tests, the software platform runs a task set of 16 periodic synthetic tasks with random parameters. Each data in the table represents the worst case result recorded in 2048 samples. The Modelsim simulator from Mentor Graphics is used to retrieve hardware scheduler results. For the software EDF scheduler, large overhead is found in task release (maximum 1079 cycles) and task activate (maximum 1427 cycles) controls in which queue insertion may occur. On the other hand, activate and suspend controls in the MLFQ scheduler are found to be time consuming.

Table 6-1 Worst case Control Overhead (clock cycles)

Command/scheduler	HW-MLFQ	HW-EDF	HW-LST	SW-EDF	SW-MLFQ
Block/exit	3	2	2	273	273
Activate	3	3	3	1427	1735
Suspend	3	3	3	649	1511
Aperiodic task release	N/A	3+D*	3+D*	1079	N/A
Periodic task release	N/A	3	3	1079	N/A
Aperiodic task complete	N/A	3	3	118	N/A
Periodic task complete	N/A	2+D*	2+D*	118	N/A

Table 6-2 Absolute Scheduling Jitter (clock cycles)

Command/scheduler	HW-MLFQ	HW-EDF	HW-LST	SW-EDF	SW-MLFQ
Block/exit	1	1	1	2	5
Activate	1	1	1	1016	1336
Suspend	1	1	1	378	783
Aperiodic task release	N/A	1+D*	1+D*	539	N/A
Periodic task release	N/A	1	1	539	N/A
Aperiodic task complete	N/A	1	1	2	N/A
Periodic task complete	N/A	D*	D*	2	N/A

The hardware scheduler generally takes one cycle to decode the control commands. Except enqueue, the internal queue controls are normally processed within two cycles. Simple queue controls completes in one cycle, while the remove control and controls that incur task switching require another cycle to reorder the queue and set the flags, for example task suspend and activate controls in MLFQ scheduler.

The MLFQ scheduler is constructed by FIFO BSR queues, thus the enqueue process only takes one cycle to complete. Meanwhile, the EDF and LST schedulers adopt a priority queue model, thus larger control overhead is found during the enqueue process. For the ASR task queue, synchronization related task controls do not trigger enqueue, and are executed within three cycles. On the other hand, the job release control causes a task's absolute deadline to be updated, and will always lead to an enqueue process. For periodic tasks, their job release time can be predicted, thus deadline calculation and task enqueue process is shifted to the time when task completes. Thus task release costs three cycles while task complete takes a maximum of 2+D cycles (D represents the depth of task queue) with the linear search. If binary search implemented, D will be replaced with log(D) +1. Note that this long control does not halt the processor and has no effect on performance.

For aperiodic tasks, the job release time cannot be predicted. Thus the enqueue process follows a job release control. It costs user 3+D cycles to complete. The job complete control is shortened to three cycles. The job release time control affects system's event response time.

Table 6-2 shows that hardware scheduler also minimize the absolute scheduling jitter related to each task control.

Maximum Clock Frequency

The additional circuitry of a hardware scheduler can potentially introduce a new critical path in the system, which could degrade system's operating frequency. Table 6-3 shows the maximum clock frequencies for the implemented hardware schedulers.

The MLFQ scheduler is only affected by the priority level, with smaller priority level having better operating frequency. Meanwhile, the depth of BSR task queue is found not to impact the clock frequency. In the new task queue model, the SCHD-INFO is not stored in queue cell, thus task queue's parallel entry gets smaller. These alleviate the bus loading problem found in the original Shift Register PQ model.

Table 6-3 Maximum Clock Frequency of Hardware Scheduler

Scheduler Configuration	Minimum Clock Period (ns)	Maximum Clock Frequency (MHz)
MLFQ (priority level 8)	5.574	179.404
MLFQ (priority level 4)	3.856	259.336
EDF (task queue depth 16)	5.088	196.541
EDF (task queue depth 32)	5.834	171.402
EDF (task queue depth 64)	6.122	163.345
LST (task queue depth 32)	5.116	195.465

Nevertheless, the queue depth affects the ASR task queue through the PMUX module. In the PMUX, both PSEL and the multiplexer circuit can be affected. PSEL is implemented using carry chain on FPGA. The queue depth influences the propagation delay. Despite that, all schedulers' maximum frequencies are faster than the clock frequency of the Leon3 processor, which runs at a maximum 129 MHz. This result argues that the implemented hardware scheduler does not degrade the overall system clock frequency. Since the paper targets small scale embedded systems, a typical scheduler is regarded to have a queue depth of 32.

Resource Usage

Table 6-4 illustrates the resource scaling for various task queue models. The Shift Register PQ is implemented based on the architecture discussed in section 3.1.2. It shows that the SR PQ will cost much more logic resources than the BSR and ASR queue. In a typical task queue with 32 queue depth and 32 bits SCHD-INFO, the resource cost in the SR PQ model is three times of the ASR PQ and four times of the BSR PQ. Especially, SR PQ's resource cost increases greatly when the SCHD-INFO bits increases. In contrast, the resource usage in BSR and ASR models only increase slightly with the SCHDINFO bit.

Table 6-4 Task Queue Resourc Cost

Queue Depth	SCHDINFO Bit Number	Shift Register PQ		BSR PQ		ASR PQ	
		LUT	Register	LUT	Register	LUT	Register
32	16	1377	768	485	327	802	404
32	32	2404	1280	501	343	818	420
32	64	4452	2304	533	375	850	452
16	32	1184	624	240	181	431	260
64	32	4811	2560	1123	700	1782	846

Table 6-5 shows the resource usage for different implementations of the hardware scheduler. The resource cost for deadline based schedulers solely depends on the task queue depth. The SCHD-INFO bits, on the other hand, does not affect resource cost except two individual BRAMs. This approach saves register cost, which is more useful for other hardware modules.

The MLFQ scheduler on the BSR model still holds the same resource problem seen on the FIFO PQ model, as discussed in section 2. When used for dynamic priority allocation algorithms, both its priority level and queue depth are required to increase along with task number, thus the resource cost will increase significantly. For a fixed priority scheduler (RM scheduling), the number of possible tasks in one priority level is usually pre-known. In such cases, the scheduler's resource usage increases linearly as priority number increases.

Table 6-5 Resource Usage for Hardware Schedulers (% of total available)

Module Name	LUT4	Registers	RAMB32
Leon3 on-chip system	6834 (9.9%)	2760 (4.0%)	N/A
MLFQ (task queue depth 8, priority level 8)	887 (1.3%)	602 (0.9%)	0
MLFQ (task queue depth 16, priority level 8)	1520 (2.2%)	1181 (1.7%)	0
MLFQ (queue depth 16, priority level 16)	3103 (4.4%)	2054 (3.0%)	0
MLFQ (task queue depth 32, priority level 8)	3065 (4.4%)	2087 (3.0%)	0
MLFQ (task queue depth 16, priority level 4)	1202 (1.7%)	690 (1.0%)	0
EDF (task queue depth 16, 32 SCHD-INFO bits)	729 (1.1%)	480 (0.7%)	2 (1.4%)
EDF (task queue depth 32, 32 SCHD-INFO bits)	1142 (1.7%)	654 (0.9%)	2 (1.4%)
EDF (task queue depth 64, 32 SCHD-INFO bits)	2170 (3.1%)	1158 (1.7%)	2 (1.4%)
LST (task queue depth 32)	1634 (2.4%)	685 (1.0%)	2 (1.4%)

System Benefit

To better illustrate the benefit of the hardware scheduler, this section compares the performance between a software MLFQ scheduler and a hardware EDF scheduler.

One critical performance metric for a real-time system is how quickly the system can respond to an external event. In many applications, events are signalled using interrupts. The relevant ISR then issues a signal to a task to perform a certain action. This paper studies a metric called the event trigger latency, which measures the worst case latency from the arrival of an interrupt until a high priority aperiodic task is triggered to run. This metric, similar to interrupt latency, represents a fundamental operation of the system and is found to be only affected by the number of tasks in the system.

The hardware scheduler can greatly reduce event trigger latency. For a system having 8 periodic tasks, the software RTOS is measured to have an event trigger latency of 8518 clock cycles. The hardware scheduler has a reduced scheduling overhead of 1424 cycles. When a hardware kernel includes mechanisms to implement time services and directly signal tasks without using an ISR [20], the event trigger latency is further reduced to 1169 cycles. The latency jitter is also reduced from 4657 to 849 cycles.

Besides response time, system workload (usually measured as processor utilization) is another critical metric. Theoretically [10], for a system having 8 periodic tasks, an EDF scheduler allows a higher processor utilization bound (100%) than a normal RM based priority scheduler (72.4%).

However, for practical systems having high task switch rates, the actual utilization available is much smaller due to scheduling and context switch overheads. The hardware scheduler, as analysed in previous sections, can remove most of the scheduling overhead, providing more computing cycles for applications.

In this paper, five task set cases are studied to find the scheduler's influence on processor utilization. Generally, all task sets have 8 artificial tasks which run with specified characteristics. The soft-core processor has a clock rate of 100MHz and system runs for a total period of 1 second.

The first three task set are almost identical, and include only periodic tasks with random generated execution time and task period (range from 500 us to 10 ms, deadline equal to period). The difference is how many jobs are released in one second. .

Two other cases are used which are drawn from real world applications. Case 4 tasks, described in Table 6-6, resemble a car body electronics application. It has two special periodic tasks, Input and Output tasks, which deal with I/O for 16 channels. After processing one channel, the task will sleep for a specified time till the next channel is ready. One aperiodic task is used to sample and process PWM signals. Case 5 simulates a car engine controller, with tasks as shown in Table 6-7. It requires strict real-time behaviour, thus many small threads are grouped into 1ms, 3ms, 5ms and 10ms tasks. Another special feature is the Sync task, which controls ignition and fuel injection. It is event triggered by the engine crank position interrupt and has the highest priority in the system. Due to the limitation of system performance, this task is simplified to have a minimum period of 1ms (rpm 1000) and 4 execution stages. It sleeps for a specified time after it completes one stage.

Table 6-6 Case 4 Task Set Characteristics

CASE4	Name	Task Description	Period	Utilization
T1 (P)	2ms	Simple periodic task	2ms	3%
T2 (P)	Monitor	Simple periodic task: task deadline check, exception handling, error mode control	10ms	1%
T3 (P)	Compute	Task activated by input task	10ms	10%
T4 (P)	Network	LIN/CAN communication	10ms	10%
T5 (P)	Service	diagnostic services (UDS, CCP), remote key	20ms	6%
T6 (P)	Input	performs 16 A/D channel sampling and input filtering	5ms	10%
T7 (P)	Output	implements 16 channel software PWM outputs	10ms	10%
T8 (AP)	PWM	Aperiodic tasks (PWM input capture & decode, triggered by timer interrupt)	2ms	2%

Table 6-7 Case 5 Task Set Characteristics

CASE5	Name	Task Characteristics	Period	Utilization
T1 (P)	1ms	Simple periodic task	1ms	5%
T2 (P)	2ms	Simple periodic task	2ms	2%
T3 (P)	5ms	Simple periodic task	5ms	4%
T4 (P)	10ms	Simple periodic task	10ms	8%
T5 (P)	Monitor	Simple periodic task, similar to case 4	10ms	1%
T6 (P)	Network	LIN/CAN communication	10ms	5%
T7 (AP)	Sync	Event triggered task, 4 execution stages	1ms	12%
T8	BG	Background Task	N/A	0~3%

Table 6-8 Processor Utilization Performance

Case No.	Job Number	Total Task Utilization	Scheduler Cost	RATIO
Case 1	1500	55.5% (max)	3.37%	6.1%
Case 2	3000	48.5% (max)	4.4%	9.1%
Case 3	5000	41.3% (max)	5.77%	14.0%
Case 4	1650	52%	5.96%	11.5%
Case 5	3050	37%	8.78%	23.7%

The experimental results (Table 6-8) show that software scheduler consumes a large portion of the processor utilization. For case 1-3, maximum utilization available for tasks is tested. Above the value, deadline miss will occur in the system. Case 5 also reaches its highest performance bound. It is found that practical task set utilization is much lower than ideal utilization bound, and degrades greatly as system's task switch rates grows higher (more jobs released each second). The RATIO column calculates the portion between scheduler and task set utilization. This shows the scheduler portion is large, ranging from 6.1% to 23.7%. The hardware scheduler can diminish this utilization cost.

7. Conclusion

The hardware scheduler operates in parallel to the processor and requires no system tick ISR when compared to a software scheduler. Thus it accelerates system's real-time performance. Due to its nature, the hardware scheduler introduces more resource cost but is acceptable as shown in above results.

This paper proposes a new task queue architecture to implement hardware schedulers for various scheduling algorithms. Compared to traditional Shift Register task queue, the proposed task queues have the following advantages:

- Enable remove control to support system monitoring and SCHD-INFO change
- Maintain a good balance between processing overhead and resource cost
- Allow an external controller to access each task's SCHD-INFO to implement complex algorithm

The main defect for this queue model is its operating frequency degrades quickly as queue depth increases. Since it is designed for task scheduler, a queue depth of 64 is enough for small scale real-time systems, which does not affect system's operating frequency.

The new task queue model helps to build more practical schedulers. It enables dynamic priority change and time slicing in the MLFQ scheduler. When constructing EDF/LST scheduler, it brings along important features including small overhead, high time resolution, deadline monitoring and long system lifetime. They also have good speed performance and the resource cost scale well for large SCHD-INFO bits.

Besides that, this research introduces a configurable scheduler architecture that fits various scheduling algorithms. The scheduler supports the configuration of task number, priority levels and RTOS interface type. This enables designers to utilize resource more efficiently and tune the scheduler to reach maximum performance for the applied application.

8. References:

[1] A. S. John and R. Rajkumar, "Real-Time Operating Systems," *Real-Time Systems,* vol. V28, pp. 237-253, 2004.

[2] T. N. B. Anh and T. Su-Lim, "Real-Time Operating Systems for Small Microcontrollers," *IEEE Micro,* vol. 29, pp. 30-45, 2009.

[3] L. Jaehwan, V. J. Mooney, III, A. Daleby, K. Ingstrom, T. Klevin, and L. Lindh, "A comparison of the RTU hardware RTOS with a hardware/software RTOS," in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference,* Kitakyushu, Japan, 2003, pp. 683-688.

[4] M. Sindhwani, T. Oliver, D. L. Maskell, and T. Srikanthan, "RTOS Acceleration Techniques - Review and Challenges," in *6th RTL Workshop,* 2004.

[5] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha, "Analysis of power dissipation in embedded systems using real-time operating systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 22, pp. 615-627, 2003.

[6] G. C. Buttazzo, *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications,* 2nd ed.: Springer, 2005.

[7] W. S. W. L. Jane, *Real-Time Systems*: Prentice Hall PTR, 2000.

[8] W. Stallings, *Operating Systems: Internals and Design Principles,* 7th ed.: Prentice Hall, 2012.

[9] A. J. Massa, *Embedded Software Development with eCos*: Prentice Hall, 2003.

[10] C. B. Giorgio, "Rate monotonic vs. EDF: judgment day," *Real-Time Systems,* vol. 29, pp. 5-26, 2005.

[11] V. Melissa, O. Luciano, A. M. M. Cesar, R. Carlos, and H. Fabiano, "RTOS Scheduler Implementation in Hardware and Software for Real Time Applications," in *Proceedings of the Seventeenth IEEE International Workshop on Rapid System Prototyping*, 2006, pp. 163-168.

[12] P. Kuacharoen, M. A. Shalan, and V. J. Mooney, III, "A configurable hardware scheduler for real-time systems," in *International Conference on Engineering of Reconfigurable Systems and Algorithms - ERSA'03*, Las Vegas, NV, USA, 2003, pp. 95-101.

[13] J. A. Stankovic and K. Ramamritham, "The Spring kernel: A new paradigm for real-time systems," *IEEE Software,* vol. 8, pp. 62-72, 1991.

[14] C. Matjazc, V. Domen, G. Roman, and A. H. Wolfgang, "Implementation of Hard Real-Time Embedded Control Systems," *Real-Time Systems,* vol. V14, pp. 293-310, 1998.

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

[16] W. A. Halang, "State of the art and open research topics in embedded hard real-time systems design," *Journal of Computing and Information Technology - CIT,* vol. 8, pp. 167-180, 2000.

[17] N. Gupta, S. K. Mandal, J. Malave, A. Mandal, and R. N. Mahapatra, "A Hardware Scheduler for Real Time Multiprocessor System on Chip," in *23rd International Conference on VLSI Design, 2010 (VLSID '10)*, 2010, pp. 264-269.

[18] M. Sung-Whan, J. Rexford, and K. G. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Transactions on Computers,* vol. 49, pp. 1215-1227, 2000.

[19] J. Adomat, J. Furunas, L. Lindh, and J. Starner, "Real-time kernel in hardware RTU: a step towards deterministic and high-performance real-time systems," L'Aquila, Italy, 1996, pp. 164-168.

[20] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass, "hthreads: a hardware/software co-designed multithreaded RTOS kernel," in *10th IEEE International Conference on Emerging Technologies and Factory Automation*, Catania, Italy, 2005, pp. 331-338.

[21] S. Saez, J. Vila, A. Crespo, and A. Garcia, "A hardware scheduler for complex real-time systems," in *Proceedings of the IEEE International Symposium on Industrial Electronics*, 1999, vol. 1, pp. 43-48.

[22] G. Bloom, G. Parmer, B. Narahari, and R. Simha, "Shared hardware data structures for hard real-time systems," in *Proceedings of the tenth ACM international conference on Embedded software*, Tampere, Finland, 2012, pp. 133-142.

[23] J. Hildebrandt, F. Golatowski, and D. Timmermann, "Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems," in *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, 1999, pp. 208-215.

[24] A. García, S. Sáez, J. Vila, and A. Crespo, "A binary-tree architecture for scheduling real-time systems with hard and soft tasks," in *Proceedings of the XIIth conference on Integrated circuits and systems design*, Natal, RN, Brazil, 1999, pp. 78-81.

[25] T. F. Oliver, S. Mohammed, N. M. Krishna, and D. L. Maskell, "Accelerating an embedded RTOS in a SoPC platform," in *Proceedings of the annual technical conference of the IEEE Region 10 (TENCON)*, 2004, vol. D, pp. 415-418.

[26] M. Molnar, "The EDF scheduler implementation in RTEMS Operating System," Diploma thesis, Czech Technical University in Prague, 2006.

[27] Xilinx. (2010). Virtex-5 FPGA User Guide. [Online]. Available: http://www.xilinx.com/.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed.: The MIT Press, 2009.

[29] J. Gaisler, S. Habinc, and E. Catovic. (2006). GRLIB IP Library User's Manual. [Online]. Available: http://www.gaisler.com