A Configurable Hardware Scheduler for Real-Time Systems

Pramote Kuacharoen, Mohamed A. Shalan and Vincent J. Mooney III
Center for Research on Embedded Systems and Technology
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332, USA
{pramote, shalan, mooney}@ece.gatech.edu

Abstract

Many real-time applications require a high-resolution time tick in order to work properly. However, supporting a high-resolution time tick imposes a very high overhead on the system. In addition, such systems may need to change scheduling discipline from time to time to satisfy some user requirements such as Quality of Service (QoS). The dynamic changing of the scheduling discipline is usually associated with delays during which some deadlines might be missed.

In this paper, we present a configurable hardware scheduler architecture which minimizes the processor time wasted by the scheduler and time-tick processing. The hardware scheduler is flexible and provides three scheduling disciplines: priority-based, rate monotonic and earliest deadline first. The scheduler in hardware also provides accurate timing. The scheduling mode can be changed at runtime, providing support for a wide range of applications on the same device. The hardware scheduler is provided in the form of an Intellectual Property (IP) block that can be customized according to the designer's input, to suite a certain application, by a tool we have developed.

Keywords: configurable hardware scheduler, hardware scheduler, real-time systems, real-time operating system, scheduling algorithm.

1. Introduction

A Real-Time Operating System (RTOS) allows real-time applications to be designed and expanded easily. However, the RTOS introduces overhead, which may prevent some real-time systems, such as high-speed packet switches, from working efficiently. As a result, deadlines may be missed. The overhead can be reduced by migrating kernel services such as scheduling, time tick (a periodic interrupt to keep track of time during which the scheduler makes a decision) processing [7], and interrupt handling to hardware. This will significantly improve the response time and the interrupt latency, provide accurate timing, and increase the CPU utilization.

An implementation of a hardware scheduler usually can support only one scheduling algorithm. Consequently, the hardware can support a narrow range of applications, which work well under the same scheduling algorithm. Unlike software components, a hardware unit is less flexible and more difficult to modify after implementation. As a result, hardware solutions are frequently avoided. However, if the hardware scheduler is configurable to support several scheduling algorithms, then the hardware solutions become more flexible.

Future embedded devices will support a wide range of applications. The hardware scheduler may need to be reconfigured at the time of application switching. For example, suppose the current application on a handheld device is running under a priority-based scheduling algorithm and suppose that the user presses a button to switch to another application, which works well under an Earliest-Deadline-First (EDF) algorithm. In order to support the new application efficiently, the hardware scheduler will be reconfigured from the priority-based mode to the EDF mode. Furthermore, different classes of applications will have different numbers of tasks in the system. Once the hardware scheduler is fabricated or configured into a Field Programmable Gate Array (FPGA), the maximum number of tasks is fixed. Therefore, the number of tasks must be specified for the application class before the hardware is built. However, the operations of the hardware scheduler should be independent of the number of tasks. Scalability of the hardware scheduler can be accomplished by implementing fixed-cycle operations. Each operation requires a fixed number of cycles. The ready queue architecture must be scalable. When the ready task is inserted to the ready queue, it must be sorted in a constant time.

Some FPGA vendors have recently released reconfigurable logic with processors such as PowerPC [13] and ARM [16]. With chips available containing both reconfigurable logic and processor(s) together on one die, the

hardware scheduler can be easily configured. Furthermore, with a runtime support environment for reconfigurable systems, any scheduling algorithm or any RTOS component implemented in hardware can be downloaded and reconfigured at runtime. This will enable the hardware solution to be as flexible as the software solution; for example, an existing part, the Xilinx XC3000 is reconfigurable in 1.5 ms, and future FPGA products promise to be reconfigurable in much less time than this [12].

We implement a configurable hardware scheduler in the Verilog Hardware Description Language (HDL) and an RTOS in C. Our implementation is scalable. We migrated the software scheduler and the time tick background processing to the hardware. Therefore, the software overhead from these services is eliminated.

The paper consists of seven sections. The next section, Section 2, describes related work in the area of scheduling algorithms implemented in hardware. In the third and the fourth sections, the configurable hardware scheduler architecture and software support are presented. In the fifth section, we discuss automatic customization of the hardware scheduler. In the sixth section, experiments and results are discussed. Finally, the seventh section concludes the paper.

2. Related work

Several previous papers deal with scheduling algorithms implemented in hardware. Most of them are in the field of packet scheduling in real-time networks [1], [2], [8]. Scheduling in such systems is based on priorities. Therefore, a key aspect is to implement priority queues. Many hardware architectures for the queues have been proposed: binary tree comparators, FIFO queues plus a priority encoder, and a systolic array priority queue [1]. Most of the hardware proposed addresses the implementation of only one scheduling algorithm (e.g., Earliest Deadline First) [8].

In the field of real-time processing, there have been few proposals of hardware implementations. In the Spring kernel project [3], [10], a coprocessor was built to enhance the multiprocessing scheduling [9]. This coprocessor was able to perform feasibility checks and calculate a complete feasible schedule. FASTHARD [4] and FASTCHART [5] are two approaches to implement a hardware kernel for single or multiprocessor systems. The FASTCHART approach used a special purpose CPU to execute the scheduling algorithm running in parallel to the main CPU. In FASTHARD, the author implemented custom hardware in an FPGA to perform the functionalities of the priority scheduler [11].

The previous research on the hardware implementation of real-time schedulers focused only on implementing only one scheduling algorithm, thus making them inefficient and not suitable for systems where the required scheduling discipline changes during runtime. We, on the other hand, introduce a configurable scheduler that supports three scheduling disciplines. The scheduler can switch from one scheduling discipline to another on the fly during runtime to adapt to changes in the system. Our hardware scheduler was designed to support multiple scheduling disciplines using minimum area overhead. The implemented scheduling disciplines share the same hardware components and use the maximum amount of common logic and minimum amount of multiplexers to select a scheduling discipline. Our implementation is entirely different from having three independent hardware schedulers running in parallel.

3. Configurable Scheduler Hardware

A programmable hardware system is designed to handle the scheduling of tasks in complex systems. The goal of the hardware design is to minimize the processor time wasted by the scheduler and by interrupt handling. Historically, designers have avoided hardware solutions because they have been considered to be inflexible and hard to modify after implementation in contrast to software solutions. However, with recent FPGA technology, this is no longer the case, with hardware reconfigurable in 1.5 ms and less (e.g., hundred of microseconds) [12]. Therefore, in this paper we take advantage of advances in FPGA technology by placing part of an RTOS in hardware, reducing, for example, scheduling and time-tick processing by thousands of assembly instructions (executing in tens of thousands of clock cycles if there are cache misses) for a system with 50 tasks.

The hardware scheduler provides three different types of scheduling algorithms: Priority (PI), Earliest Deadline First (EDF), and Rate Monotonic (RM). Also, the hardware scheduler supports preemption at the scheduler level and at the process level. The hardware scheduler supports up to eight levels of interrupts and provides accurate timing. The hardware was designed to minimize the processor overhead while maintaining flexibility and extensibility. In the following section, we will describe the hardware scheduler architecture, commands and interfacing.

3.1. Architecture

The proposed architecture for the hardware scheduler is shown in Figure 1. The main components of the scheduler are:

- The Sleep Queue (SQ),
- The Priority Queue (PQ),
- The Task Table,
- The Interrupt Controller and
- The Control Unit

which will be described in the following sections.

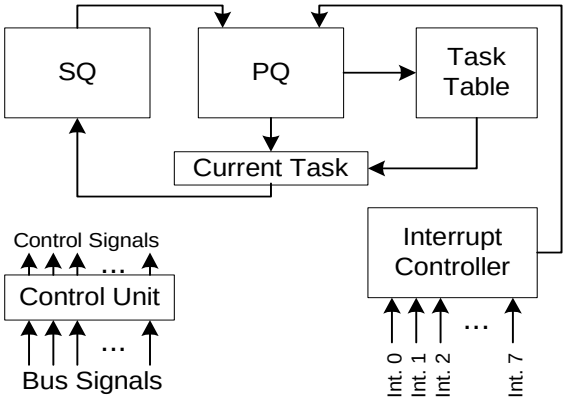

Figure 1. The configurable hardware scheduler micro-architecture.

3.1.1. Priority Queue (PQ)

The priority queue is a sorted queue used to store the active tasks in a sorted order (ready queue). The queue entry is shown in Figure 2. The REG field is a 32-bit register that is used to hold either the priority in the case of a priority-based scheduler or the period in the case of an RM scheduler. The counter field holds either the period for RM or priority-based schedulers, or the time to the deadline for an EDF scheduler. The queue can be sorted according to either the REG field in the priority-based or RM scheduler mode or the counter field in EDF scheduler mode.

Figure 2. The PQ entry format.

We are using a priority queue very similar to the priority queue described in [8]. When a task is inserted, the queue automatically re-orders itself. Figure 3 shows the architecture of the basic cell of the queue.

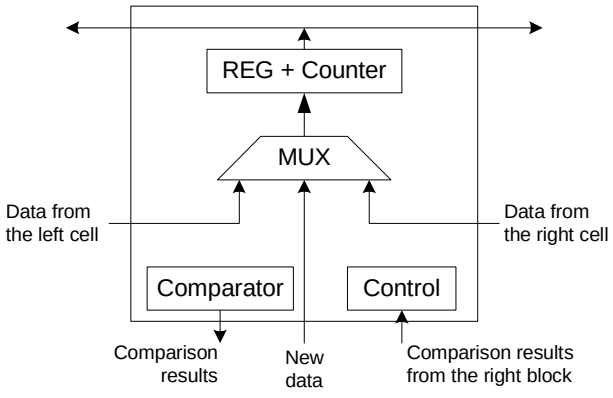

Figure 3. The PQ cell architecture.

Each cell consists of a storage element, a multiplexer, a comparator and control logic. During the en-queue operation, the new entry is broadcast to all the cells. Each cell makes a local decision as to what action to take, with only one of the cells latching the new entry. The others will either keep their current entry or latch the right neighbor's entry. The net effect is to have the new entry force all entries with lower priority to shift one cell to the left, while the new entry places itself to the left of the entries with higher and equal priority. A de-queue operation shifts all entries one cell to the right. The insertion and the ordering process takes only one clock cycle in all cases [8].

3.1.2. Sleep Queue (SQ)

The sleep queue is used to hold the sleeping tasks, either by issuing the SLEEP or YIELD commands. The sleep queue uses an architecture similar to that of the PQ. However, the SQ entries are sorted according to their sleep time, specified by the SLEEP command or the remaining time to the end of the period when the YIELD command is issued. Figure 4 shows the data format of the SQ entry.

Figure 4. The SQ entry format.

3.1.3. Task Table

The Task Table is a lookup table indexed by the task ID. The format of the entry is shown in Figure 5: the PRI, Period, and WCET fields are used to hold the task priority, period, and worst-case execution time, respectively. The TYPE field is used to hold the task type: periodic or aperiodic. The PRE field indicates if the task can be preempted by other tasks. The STATUS field holds the task status: active, suspended, or deleted. Every time a task is activated, the scheduler fetches the task information from the task table.

Figure 5. The task table entry format.

3.1.4. Interrupt Controller

This module is used to handle external interrupts. The module supports up to eight interrupt levels. Each interrupt can be assigned to a task to handle the associated interrupt level. Each interrupt can be configured to be either fast interrupt (the interrupt handling task will run right away by preempting the current task) or slow interrupts (the handling task will be inserted to the PQ).

3.1.5. Control Unit

The control unit is used to interface the hardware scheduler to the external host. The control unit accepts a command, decodes the command and generates proper control signals to the rest of the hardware to execute the command.

3.2. Hardware Scheduler Commands

The hardware scheduler implements the time-tick handling and the execution of the chosen scheduling algorithm, while the context switching is done in software. The hardware scheduler has a set of commands to allow the software portion to configure the hardware and to perform operations. The commands are issued through a memory mapped I/O port, which can be done in one or two clock cycles depending on the size of the command word. For example, since the SLEEP command uses 32 bits for the sleep time, it uses two words (64 bits) overall and thus takes two clock cycles to execute. The SSLEEP (Short SLEEP) command, on the other hand, uses 22 bits for the sleep time and can fit the overall command in 32 bits; thus, SSLEEP can execute in one clock cycle. Table 1 lists the commands that can be executed by the hardware scheduler.

Table 1. Hardware Scheduler Commands.

	Command	# of Cycles
Scheduler Related	STOP	1
	RUN	1
	CONFIGURE	1
Task Related	CREATE Task	1
	MODIFY Task	2
	SLEEP	2
	SSLEEP	1
	YIELD	1
	SUSPEND	1
	RESUME	1
	DELETE	1

These commands are standard RTOS task creation/deletion and scheduling APIs. The STOP, RUN and CONFIGURE commands are used for disabling, enabling and configuring the hardware scheduler. The CREATE command creates a new task. The task's parameters (e.g., task priority and task worst-case execution time) can be modified using the MODIFY command. To delay a task, SLEEP or SSLEEP can be used. The YIELD command will insert a task to the SQ for the remaining time in the period. The SUSPEND command suspends a task while the RESUME command resumes a suspended task. A task can be deleted using the DELETE command.

Example 1: Consider a 32-bit system that utilizes the hardware scheduler which is configured to work in a priority scheduling mode and supports up to 64 tasks. The time tick resolution for schedule is set to 10 μs. To create a task, the real-time operating system must issue the CREATE command which requires the task ID and the task priority. The CREATE command is a 32 bit command where the task id and the task priority occupy 6 bits each. Therefore, the CREATE command can be issued in one cycle. One of the tasks is a periodic task which reads an input every 45 s. The task needs to idle (sleep) for 45 s after reading each input value. In order to idle, the task calls an API function that utilizes the SLEEP and SSLEEP commands. Since 45 s are equivalent 4.5 million ticks which need more than 22 bits to be represented, the API call uses the SLEEP command which takes two cycles to execute. □

3.3. Hardware Scheduler Interfacing

The hardware is designed to be able to interface easily with any microprocessor. The hardware scheduler can be connected to a bus to act as a memory mapped port, or it may be connected to the processor as a co-processor. In addition, if the processor (such as the StarCore SC140 DSP core [14]) supports instruction-set accelerators, the hardware scheduler can be used to extend the processor instruction-set to manage the system processes with customized assembly instructions such as the YIELD and RESUME commands explained in Section 3.2.

Figure 6 shows the hardware scheduler connected to a processor as an I/O port.

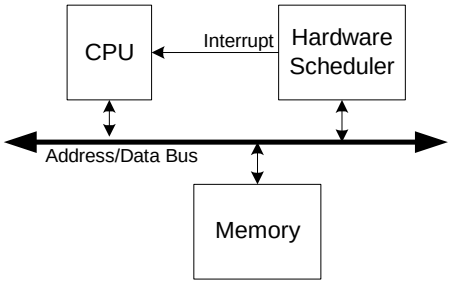

Figure 6. The hardware scheduler connected as an I/O device.

In this configuration, the hardware scheduler has one address to which the commands can be written and from which the status can be read. The hardware scheduler directs the processor to switch to another task when a higher priority task is ready by sending an interrupt signal to the CPU. When the CPU is interrupted, it transfers the control to the context switcher, which reads the task ID from the hardware scheduler, stores the context of the current task, and switches the context to the task with the ID read from the hardware.

4. Software Support

The RTOS consists of processor-independent code and processor-specific code. Therefore, the RTOS for the hardware scheduler can be easily ported by modifying the

processor-specific code. Since the hardware scheduler cannot directly access the registers of the processor, the context switching is done in software. During context switching, the contents of the registers are stored in the stack of the current task, and the contents of the registers of the new task are restored. The context switching time depends on the number of registers of the processor. The APIs of the hardware scheduler are provided as the kernel services. The following steps show a pseudo code for a typical application: (a) configure scheduler, (b) initialize the RTOS, (c) create tasks and (d) start multitasking.

When the multitasking is started, the hardware scheduler schedules tasks. It interrupts the RTOS to perform a context switching to run the first task.

5. Automatic Customization of the Scheduler

Figure 7 gives an overview of the flow of our scheduler customization tool.

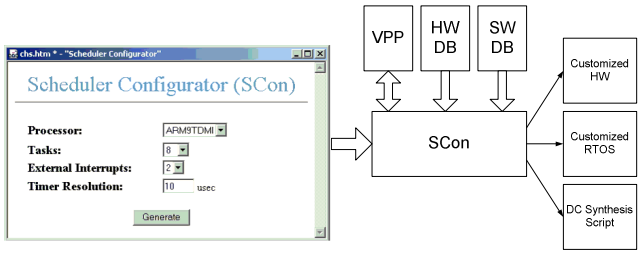

Figure 7. The Scheduler Customization Flow.

A Graphical User Interface (GUI), which consists of set of HTML forms, captures the user inputs and passes them to the scheduler customization application (developed in C-Language). We call this application **S**cheduler **C**onfigurator (SCon). SCon processes the user inputs, validates them and generates the scheduler hardware files (Verilog format) and the corresponding software that enables a RTOS to use the hardware. Also, SCon generates the necessary Verilog files (wrapper) to interface the hardware scheduler to the processor. Moreover, SCon generates Synopsys DC™ synthesis scripts for the hardware scheduler.

The following is a partial list of the user specified parameters:

- Number of tasks
- Number of external Interrupts
- Timer Resolution
- Processor Type

In order to generate the hardware files, a database of parameterized Verilog files of each system component is being used. The Verilog files in the database are written in such a way that a custom version of the file can be generated using a Verilog PreProcessor (VPP) [15].

Once the user configurations and settings are captured, SCon selects from the hardware database the suitable scheduler bus interface and the parameterized verilog files of the hardware scheduler. Next, SCon sets the parameters of each verilog file to reflect the user input. The hardware components (Verilog files) are passed to VPP which processes them and generates new customized Verilog files. Finally, SCon configures the RTOS according to the user input. The output from SCon is a set of Verilog files for the hardware, a set of C and assembly files for the RTOS and Synopsys DC synthesis script file.

6. Experiments and Results

We verified the hardware scheduler and the RTOS using hardware/software co-design tools, namely, Synopsys VCS, Mentor Graphics Seamless CVE and Mentor Graphics XRAY. VCS is used for simulating the hardware in Verilog HDL. Seamless CVE interfaces the hardware and the software simulators. XRAY is used as the instruction set simulator and debugger. We simulated a System-on-Chip (SoC) similar to that illustrated in Figure 6. The hardware scheduler is set up as an I/O device as illustrated in Figure 6. We used a PowerPC 750, with Level 1 instruction and data caches each of 32KB, as the processor which runs at 400 MHz while the bus runs at 133 MHz and can deliver a peak performance of 733 MIPS [17]. The memory size of the system is 4MB.

6.1. Scheduler Overhead

The simulation results show that for a system that utilizes the hardware scheduler, the assembly instructions executed by the scheduler and the background time tick processing are eliminated as shown in Table 2. In Table 2, the programs were compiled using the GCC cross compiler for PowerPC, and the results are in number of assembly instructions. MicroC/OS II scheduler is a priority-based scheduler [7]. For time-tick processing, the RTOS periodically checks every task and decrements the delay value if it is not zero. The upper bound of the processing time is directly proportional to number of tasks in the system. This overhead is large if there are many tasks and the time tick resolution is high. As a result, the CPU utilization is reduced, and tasks may miss their deadlines.

Table 2. The assembly instruction execution comparison between Micro-C/OS II and the hardware scheduler.

	Micro C/OS II	Hardware Scheduler
Scheduler	69	0
Time-tick processing	47+47*(number of tasks)	0

Figure 8 shows the overhead percentage (percentage of CPU time spent processing the time ticks) as a function of the time tick resolution. Figure 8 shows that for a system with 32 processes (tasks) and a time tick of 1 ms, 0.21% of the CPU time is wasted (i.e., used for time tick processing).

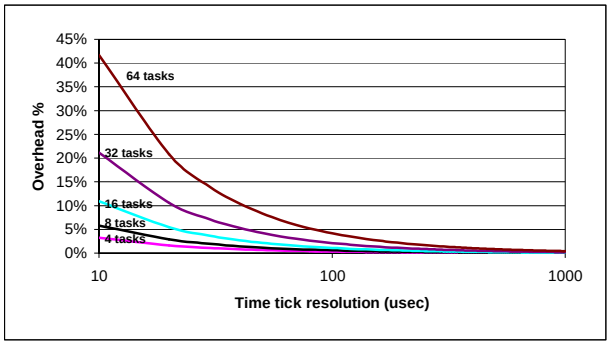

Figure 8. The scheduler and the time tick processing overheads in MicroC/OS II.

However, if the time tick resolution becomes 10 us, 21.16% of the CPU time is wasted. Since the hardware scheduler eliminates such overheads, the response time and the interrupt latency are improved. The appropriate task can be executed when the hardware scheduler sends an interrupt to the processor. If the system has a fast clock so that the 21.16% overhead does not make the system miss any deadlines, the introduction of the hardware scheduler would make the system run at a clock speed that is 21.16% less. A reduced clock frequency allows a lower core voltage which results in a reduction of the processor power consumption (please note that the power consumption of the hardware scheduler is negligible when compared to the processor power consumption since the hardware scheduler occupies far less area – see Table 4 – than the processor and has much less transistor switching activity).

Example 2. It is likely that next generation handheld devices will support multiple applications such as wireless communication and a Voice User Interface (VUI). These applications may work well under different scheduling algorithms. For example, the wireless communication application may work well under an EDF scheduler, and the VUI may work well under a priority-based scheduler. If the handheld device has only one scheduling algorithm, it cannot efficiently support both applications. However, if the handheld device has a configurable hardware scheduler, multiple applications can select the scheduling algorithm, which fits their requirements. □

In this experiment, we simulate the scheduler for such a handheld device using Seamless CVE. Initially, the handheld device is running a VUI application using a priority-based scheduling algorithm. When the user switches to a wireless communication application, the VUI application must be suspended. The software sends command to the hardware scheduler to suspend the VUI application, to configure the hardware scheduler to operate in the EDF mode, and to create tasks for the wireless communication application.

Table 3. Number of PowerPC assembly instructions of the APIs.

API	# of PPC Assembly Instructions	WCET (# of cycles)
configureScheduler	37	230
SuspendTask	21	125

The application-switching overhead is shown in Table 3. The values in Table 3 are in number of PowerPC assembly instructions. The actual commands sent to the hardware scheduler are one PowerPC assembly instruction for suspending a task and for configuring the hardware scheduler, and three PowerPC assembly instructions for creating a task. Moreover, each API has less assembly instructions than the context switching routine. This dynamic change of the scheduler at runtime is not supported by most commercial RTOSes. Furthermore, even if a software RTOS were to support such dynamic changing of the scheduler, such a software RTOS would be an order of magnitude or more slower (especially considering WCET cache behavior) in changing the scheduler. Table 3 assumes that cache misses take at most eight cycles to fill a cache line.

In our case, due to limited memory in the handheld device, the software RTOS schedule change causes the memory buffer in the handheld device to overflow, whereas the speedy hardware scheduler change takes effect before the memory overflows. Furthermore, during actual operation, the software RTOS would cause some timing constraints to be missed while the hardware RTOS allows all timing constraints to be met, especially when considering the memory interface.

6.2. The Hardware Scheduler Synthesis Results

We developed a RTL Verilog model for the hardware scheduler. As illustrated in Table 4, we synthesized the hardware scheduler for the HP 0.35µ process. The synthesized hardware supports up to 16 tasks and up to eight external interrupt sources. The hardware scheduler uses 1115 standard cells and occupied an area of 0.24 mm^2.

Table 4. Synthesis result using HP 0.35µ process.

Number of standard cells	Area (mm^2)
1115	0.24

Table 5 shows the synthesis result using Altera Quartus II for the EP20K family. The hardware scheduler uses 421 logic elements and 564 registers.

Table 5. Synthesis result using Altera Quartus II for EP20K

Number of Logic Elements	Number of Registers
421	564

7. Conclusion

We implemented a configurable hardware scheduler and a real-time operating system. Both components are verified in a hardware/software co-design environment. The configurable hardware scheduler is flexible; it supports three scheduling algorithms, namely, priority-based, rate monotonic, and earliest-deadline-first. The scheduling and the time-tick processing overhead are eliminated from the real-time operating system. Also, we presented a tool that can customize the hardware scheduler to suite a particular system.

8. Acknowledgements

This research is funded by the State of Georgia under the Yamacraw initiative and by NSF under INT-9973120, CCR-9984808 and CCR-0082164. We acknowledge donations received from Denali, Hewlett-Packard Company, Intel Corporation, LEDA, Mentor Graphics Corp., SUN Microsystems and Synopsys, Inc.

9. References

[1] S. Moon, J. Rexford and K. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Transactions on Computer*, vol. 49, no.11, pp.1215 –1227, November 2000.

[2] D. Picker and R. Fellman, "A VLSI priority packet queue with inheritance and overwrite," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3 no. 2, pp. 245–253, June 1995.

[3] J. Stankovic, D. Niehaus and K. Ramamritham, "SpringNet: A Scalable Architecture for High Performance, Predictable and Distributed Real-Time Computing," University of Massachusetts, Amherst, Massachusetts, Tech. Rep. UM-CS-1991-074, 1991.

[4] L. Lindh, "FASTHARD – A Fast Time Deterministic Hardware Based Real-Time Kernel," *Proceedings of the Fourth Euromicro Workshop on Real-Time Systems*, pp. 21-25, June 1992.

[5] L. Lindh and F. Stanischewski, "FASTCHART - Idea and Implementation," *Proceedings of the International Conference on Computer Design (ICCD)*, pp. 401-404, January 1991.

[6] J. A. Stankovic et al., *Deadline Scheduling for Real-Time Systems – EDF and Related Algorithms*, Kulwer Academic Publications, New York, 1998.

[7] Jean Labrosse, *Micro C/OS: Real Time Kernal II: The Real-Time Kernel*, R&D Books, Kansas, 1998.

[8] B. Kim and K. Shin, "Scalable hardware earliest-deadline-first scheduler for ATM switching networks," *Proceedings of the Real-time Systems Symposium*, pp. 210-218, December 1997.

[9] W. Burleson et al., "The Spring Scheduling Co-Processor: A Scheduling Accelerator," *Proceedings of the International Conference on Computer Design (ICCD)*, pp. 140-144, October 1993.

[10] J. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Systems," *IEEE Software*, vol. 8, no. 3, pp. 62-72, May 1991.

[11] J. Adomat et al., "Real-Time Kernel in Hardware RTU: A Step towards Deterministic and High-Performance Real-Time Systems," *Proceedings of the 1996 Euromicro Workshop on Real-Time Systems*, pp. 164-168, June 1996.

[12] Xilinx, "Dynamic Reconfiguration," Application Note, 1997, http://www.xilinx.com/xapp/xapp093.pdf

[13] Xilinx Vertex-II Pro platform, http://www.xilinx.com/virtex2pro

[14] Starcore, SC140 DSP Core Reference Manual, http://e-www.motorola.com/brdata/PDFDB/docs/MNSC140CORE.pdf

[15] Verilog PreProcessor, http://www.surefirev.com/vpp/

[16] Altera Excalibur, http://www.altera.com/products/devices/arm/arm-index.html

[17] MPC750 Fact sheet, http://e-www.motorola.com/brdata/PDFDB/docs/MPC750FACT.pdf