HyOS: A Hybrid Operating System Design Approach for Real-Time Systems using Hardware Acceleration

Insop Song
Ericsson Inc.
200 Holger Way, San Jose, CA
insop.song@ericsson.com

Abstract

In this research, a hybrid hardware/software operating system design approach is presented for improving predictability and reducing overhead. A hardware priority queue is implemented and interfaced with Linux operating system process scheduler. Implementing a hardware priority queue demonstrates the consistent response time and shorter response time. A benefit of implementing a generic data structure with hardware and integrating with software as a hybrid approach is described. A field programmable gate array (FPGA) is used for implementing Hybrid Operating System (HyOS) to holds a soft-core microprocessor and a hardware priority queue. We implement Hybrid OS (HyOS) with Linux as an operating system and MicroBlaze as a soft-core microprocessor. This research suggests and demonstrated the idea that identifying generic software components within operating systems and realize them by hardware accelerations could improve system performance and easier be accepted by industry.

1 Introduction

Predictability in real-time system is one the most important aspects and a criterion for real-time operating systems is to provide a predictable scheduler to meet time constraints. In this research, a hybrid hardware/software operating system design approach is presented to increase predictability of the operating system scheduler and reduce the overhead of operating system software.

In most operating systems, a process scheduler runs most frequently and its critical process is to decide which process to schedule next, *i.e.* prioritizing runnable processes. Except few simple operating systems, the selection of a next process to run requires non-trivial computation. The computation complexity and time varies depends on the system load or number of runnable processes. Therefore, the process scheduler is one of the busiest software components in operating system; hence, it contributes to the overhead and response jitters.

In operating system, a queue or priority queue is used in many components, such as process scheduling, event management, and time management. Linux, operating system used in many embedded systems, uses a Red-black tree priority queue to sort processes based on the scheduling key (priority, fairness, or deadline). Red-black tree is also used in high-resolution timer [17] to sort based on expiration time. Other scheduler class also uses Red-black tree for priotizing the process entries [18]. Even a small performance improvement on these priority queues would improve overall system performance.

Implementing co-processor and hardware accelerator is a common approach in many application domains when there is a specific task that could be hardware off-loaded. Few examples include math co-processor, graphic processing, and network packet processors.

In this research, a hardware priority queue, which is called Tagged up/down sorter, is implemented and interfaced with Linux process scheduler software. Implementing a hardware priority queue guarantees the consistent response time and shorter response time then software implementation. A Field Programmable Gate Array (FPGA) is used for implementing Hybrid Operating System (HyOS) to holds a soft-core microprocessor and a hardware priority queue. We implement HyOS with Linux as an operating system and MicroBlaze as a soft-core microprocessor.

This approach demonstrates that Hybrid Operating System reduces computation time in finding next process and maintains consistent response time compared to software only operating system.

Organization of the paper is as follows: we present an overview of existing hardware-assisted operating system research. Next we introduce the implementation of Hybrid operating system with Linux, and then show the experimental result. Finally, summary and further research topic are described.

2 Overview of Hardware Operating System

In this section, overview of hardware operating system is described. Main approaches are to implement task management & scheduling and event management in hardware in order to reduce the overhead and minimize jitters.

FASTCHART designed operating system scheduler modules using hardware [3]. The Real-Time Unit (RTU) in it can manage 64 tasks with 8 priorities. It allows predictable real-time system using hardware OS modules. This work later evolved to a commercial hardware operating system called Sierra [2], which runs with soft-core CPU on the FPGA. Sierra use its custom software as an system call helper functions and also demonstrated with other software OS as scheduler plug-in [4].

One of the early work that worthwhile to mention is Silicon TRON [12]. Authors implemented most of the TRON OS system calls in hardware, and this work is implemented in VSLI chip.

F-Timer is a dedicated hardware unit that processes 32 tasks list organized by time priority [11]. It consists of task processing and interrupts processing modules. The design is implemented using FPGA.

Morton and Loucks also implemented entire OS with hardware [9]. Uniprocessor Earliest Deadline First (EDF) scheduler is designed and implemented with soft-core FPGA processor. Authors also show the comparisons between software only OS and hardware designed OS.

`hthreads` is a hardware and software co-designed multi-threaded OS kernel [10]. Authors provided hybrid threaded programming model using both software and hardware. In order to minimize jitters, both scheduler processing and the state management migrated to hardware from software.

In [13], authors implemented task scheduling, time management, and event management with hardware. They use a cycle-accurate simulator to evaluate their hardware OS with MicroC/OS-II [14].

3 Hybrid Operating System

In many research projects shown in previous subsection, an operating system is implanted by hardware. The advantages of implementing entire or partial operating system components with hardware are well demonstrated including reducing jitters and overhead. However, it is not easy to find either commercial or non-commercial hardware operating systems in practical applications yet. The followings are the reasons.

First, software part of the hardware operating system is less flexible and requires major changes to the operating system software and application software. This could be solved if the hardware modules changes are hidden to application programs; however, it usually not the case. Sierra, commercial hardware operating system, provides limited number system calls and requires porting of existing applications [2].

Second, design and fabrication of complicated hardware modules for operating system requires large investment and resource. Unless there are large demands, it is unlikely to be realized by chip manufactures. On the other hand, many hardware accelerations could be found in the areas with large demands, such as graphics and networking industry.

Modern operating systems are usually very complicated, and making the whole operating system with hardware could be almost impractical. Many software projects use existing commercial and open source components that tightly coupled with operating systems. Any change to operating system requires application software changes, which are less desirable in many software projects. This means that if hardware accerlation to be used, required changes to software should be minimized.

In this work, we implement a hardware component that replaces the software component that is frequently used in operating systems. The hardware component is integrated with existing operating system. Since we only take a generic data structure into hardware, hardware component design is less complicated and could be easily added to CPU if chip manufactures find demands and competitive edges from doing so. Also only minor changes are needed in operating system software.

3.1 Operating System scheduler

A process scheduler is a core part of the operating system and decides next process to run. In Linux, CFS (Completely Fair Scheduler) handles process scheduling and maximizes processor utilization. CFS computes the fairness value based on waiting time for the processor. CFS needs a sorter that sorts based on the waiting time as a key in order to choose the next process. Red-black tree is used for sort processes based on the key.

Red-black tree is known as an efficient data structure allows for efficient management of the containing entries. Even a small optimization on this data structure would improve the system performance since it is called quite frequently. One of the main ideas of this research is to use a hardware component instead of Red-black tree data structure to optimize operating system performance, which leads the next subsection to describe hardware sorter.

3.2 Hardware Sorter

Hardware sorter, a priority queue, is used in many application areas including network packet processors and proved to be efficient [5, 6, 7, 8]. We use a hardware based sorter called Tagged up/down sorter [5]. The tagged sorter provides two operations: insert and extract minimum (or maximum). The operation can be done within one clock cycle. It also keeps FIFO order if the key values are identical.

We replace Red-black tree in Linux with Tagged sorter to improve performance, which is described in next subsection.

List 1 shows the vhdl entity defnition for the Tagged sorter implementation used in this research.

Listing 1: Tagged sorter entity (interface port) definition from TSProject32.vhd

```
-- Tagged sorter
ENTITY TSProject32 IS
  PORT
  (
    clk : IN STD_LOGIC;
    reset : IN STD_LOGIC;
    insert : IN STD_LOGIC; -- enqueue trigger
    extract : IN STD_LOGIC; -- dequeue trigger

-- enqueuing elements (key & value)
    LIkey : IN STD_LOGIC_VECTOR ( 0 TO
         WIDTH_KEY-1 );
    LIdata : IN STD_LOGIC_VECTOR ( 0 TO
         WIDTH_DATA-1 );

-- dequeuing elements (key & value)
    ROkey : BUFFER STD_LOGIC_VECTOR ( 0 TO
         WIDTH_KEY-1 );
    ROdata : BUFFER STD_LOGIC_VECTOR ( 0 TO
         WIDTH_DATA-1 );

    Q_FULL : OUT STD_LOGIC;
    Q_EMPTY : OUT STD_LOGIC
  );

END TSProject32;
```

3.3 Hybrid OS design using hardware component

We use FPGA to implement Hybrid OS, which includes CPU and Tagged sorter. A soft-core microprocessor, MicroBlaze, and hardware sorter is connected through PLB (processor local bus) bus shown in Figure 1. Tagged sorter sorts using the same way as Red-black tree and returns minimum key entries to CFS scheduler.

Then kernel scheduler code is modified to use the hardware Tagged sorter. Instead of enqueue/dequeue to Red-black tree, the kernel scheduler enqueues and dequeues to the hardware sorter. Only few lines of code change are required, and no application code change is necessary.

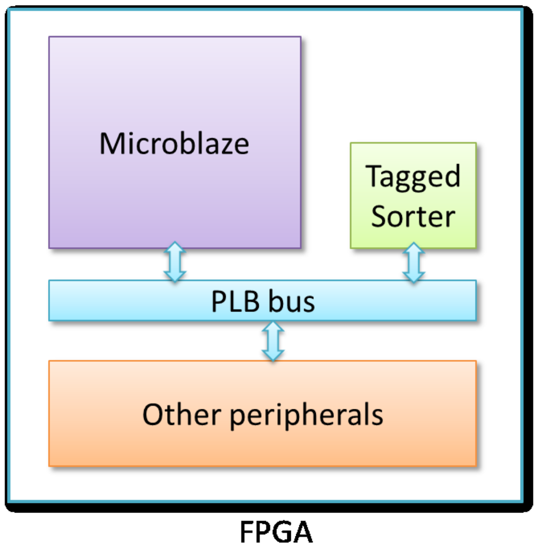

FIGURE 1: *Hybrid OS with FPGA soft-core processor and tagged sorter*

List 2 shows cleaned code fragment of enqueuing to tagged sorter extracted from `kernel/sched_fair.c`. In `__enqueue_entity()` of CFS scheduler, it calls `__enqueue_tagged_sorter()` to enqueue to Tagged sorter hardware instead of Red-black tree.

Listing 2: Hybrid OS enqueue code fragment from modified sched_fair.c

```
/*
 * Tagged Sorter based Priority Queue
 */
/* Definitions for peripheral HYOS_PLB_0 */
#define XPAR_HYOS_PLB_0_BASEADDR 0xC0E00000
#define XPAR_HYOS_PLB_0_HIGHADDR 0xC0E0FFFF

/*
 * enqueue entity to Tagged Sorter (HyOS)
 */
static void __enqueue_tagged_sorter(unsigned int key,
     unsigned int data)
{
  if (init == 0) {
    hyos =
         (char*)ioremap(XPAR_HYOS_PLB_0_BASEADDR,
         0x10000);
    init = 1;
  }

  if (hyos != NULL) {
    iowrite32(key,hyos+4*4); // key
    iowrite32(data,hyos+5*4); // val
    iowrite32(0xffffffff,hyos); // enqueue trigger
    iowrite32(0x0,hyos);
  }
}

/*
 * modified __enqueue_entity() to use Tagged Sorter
 * instead of rb-tree
 */
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct
     sched_entity *se)
{
  struct rb_node **link =
       &cfs_rq->tasks_timeline.rb_node;
  struct rb_node *parent = NULL;
  struct sched_entity *entry;
  struct timeval t1, t2;

  s64 key = entity_key(cfs_rq, se);
  do_gettimeofday(&t1);
  __enqueue_tagged_sorter(key, se); // enqueue to
       Tagged Sorter
  do_gettimeofday(&t2);

  if (printk_ratelimit())
    printk("_enqueue_cfs: delta %lu\n", t2.tv_usec -
         t1.tv_usec);
}
```

4 Experimental result

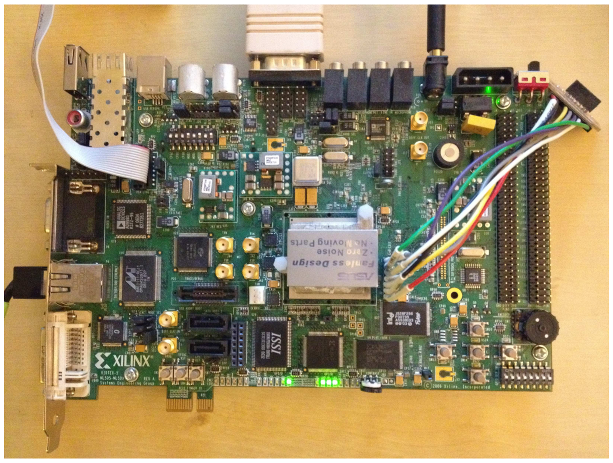

FIGURE 2: *Xilinx ML505 reference board used in this research*

This section presents the experimental results including simulation and real hardware testing. Xilinx Virtex5 (xc5vlx50tff1136-1) FPGA is used for the experiment, which is included in Xilinx ML505 evaluation board shown in Figure 2.

Table 1 shows the resource (registers and LUTS) usage for implementing Tagged sorter, not including Microblaze soft-core processor resource usage.

4.1 Simulation results

Figure 3 shows the RTL simulation result using ModelSim [16]. Enqueuing order with key and value pair is as follows: (4:1), (2:4), (4:2), (1:7), (3:4), (9:4), (6:1), (7:4), (9:5), (9:6), (153:134), (409:135), (145:134), and (265:135). Note that there are entries with the same keys: 4 and 9. Dequeuing returns based minimum key entities with FIFO order preserved, which is as follows: (2:4), (3:4), (4:1), (4:2), (6:1), (7:4), (9:4), (9:5), (9:6), (145:134), (153:134), (265:135), and (409:135). Both enqueuing and dequeuing entries done within one clock cycle and it returns (-1,-1) when the queue is empty. Tagged sorter is implemented by VHDL (VHSIC hardware description language).

4.2 Experimental results

Time taken for CFS enqueue and dequeue are measured with non-modified Linux and Linux with HyOS (Hybrid OS). We use *yes* program for simulating processor load. 2.6.37 kernel is used for the test.

Table 1: FPGA device utilization for Tagged sorter

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	8,352	28,800	29%
Number of Slice LUTs	8,910	28,800	30%

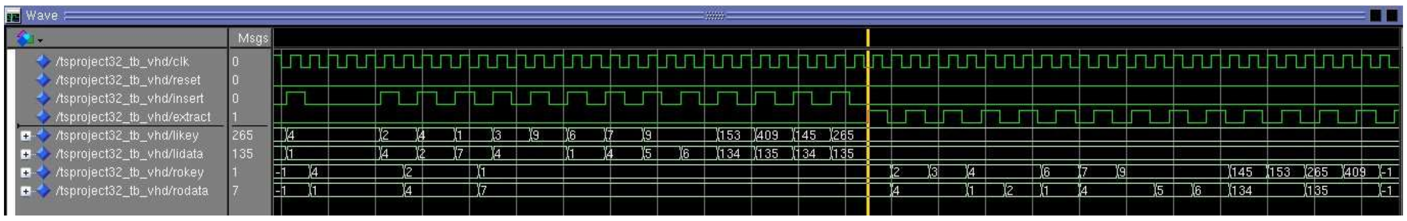
Wave
Msgs
/tsproject32_tb_vhd/clk 0
/tsproject32_tb_vhd/reset 0
/tsproject32_tb_vhd/insert 0
/tsproject32_tb_vhd/extract 1
/tsproject32_tb_vhd/likey 265
/tsproject32_tb_vhd/lidata 135
/tsproject32_tb_vhd/rokey 1
/tsproject32_tb_vhd/rodata 7

FIGURE 3: *Simulation result of enqueue and dequeue of Tagged sorter*

Figure 4 shows the enqueing time difference between software OS and Hybrid OS when no "yes" is running. Enqueue time of software is about twice longer then Hybrid OS approach. y axis unit is micro second.

Figure 5 shows the dequeuing time difference between software OS and Hybrid OS when no "yes" is running, which shows similar values as in Figure 4.

Figure 6 shows the enqueuing time difference between software OS and Hybrid OS when four "yes" are running. Hybrid OS output shows the same as when there was no "yes" (no processor load); however, software OS values are increased and shows more fluctuations.

Figure 7 shows the dequeuing time difference between software OS and hybrid OS when four "yes" are running, which shows similar trends as in Figure 6.

Figure 8 and 9 show the enqueue and dequeue time increment of software OS from no "yes" to four "yes". Time taken is clearly increased.

On the other hand for the Hybrid OS, Figure 10 and 11 show the enqueue and dequeue time increment of Hybrid OS from no "yes" to four "yes". As is expected that time taken for enqueuing and dequeuing are nearly the same.

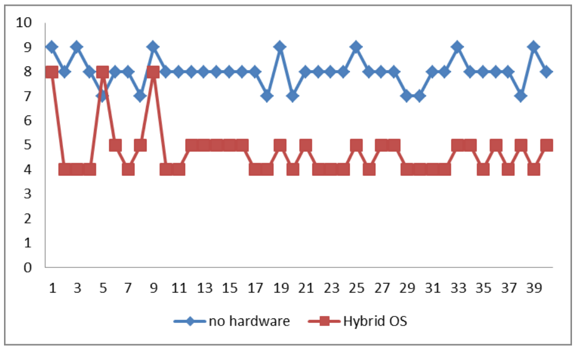

FIGURE 4: *Enqueue time comparison between software OS and Hybrid OS when no "yes" running. y-axis unit:usec*

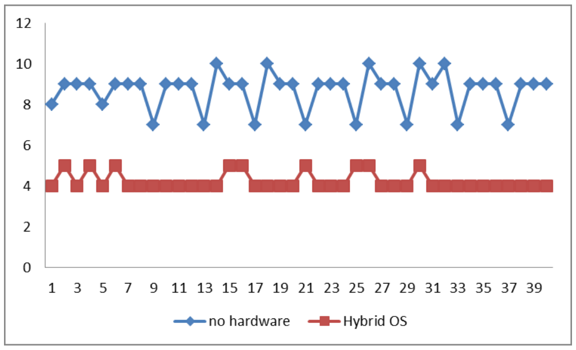

FIGURE 5: *Dequeue time comparison between software OS and Hybrid OS when no "yes" running. y-axis unit: usec*

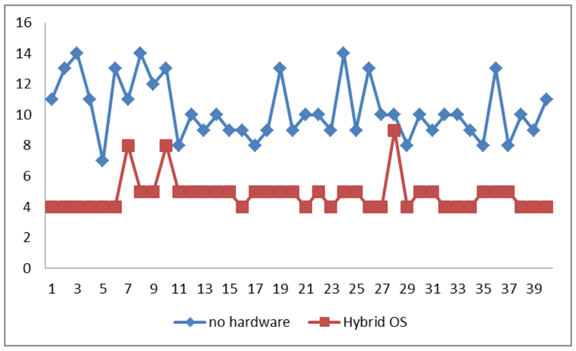

FIGURE 6: *Enqueue time comparison between software OS and Hybrid OS when four "yes" running. y-axis unit: usec*

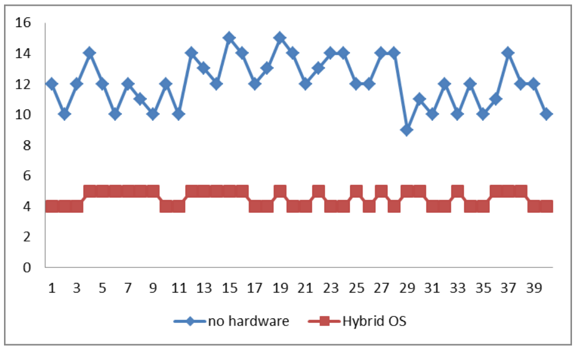

FIGURE 7: *Dequeue time comparison between software OS and Hybrid OS when four "yes" running, y-axis unit: usec*

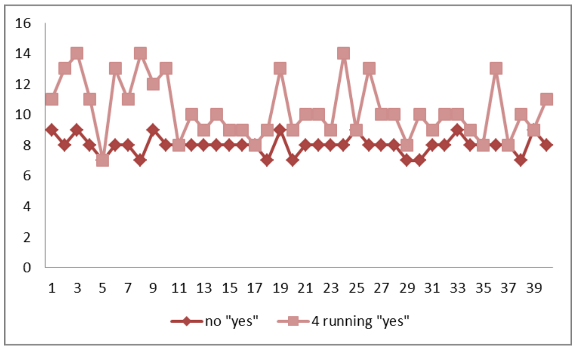

FIGURE 8: *Enqueue time comparison of software OS when no load and four "yes", y-axis unit: usec*

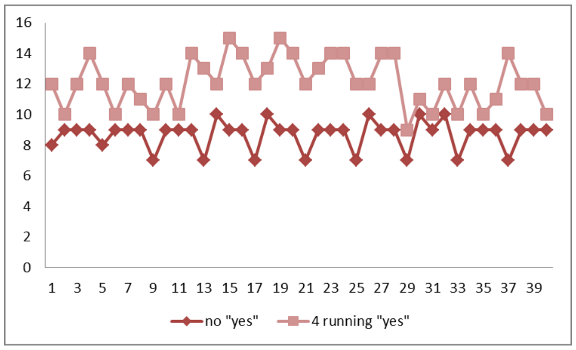

FIGURE 9: *Dequeue time comparison of software OS when no load and four "yes", y-axis unit: usec*

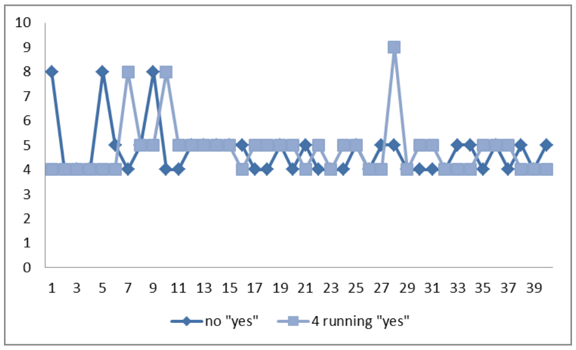

FIGURE 10: *Enqueue time comparison of Hybrid OS when no load and four "yes", y-axis unit: usec*

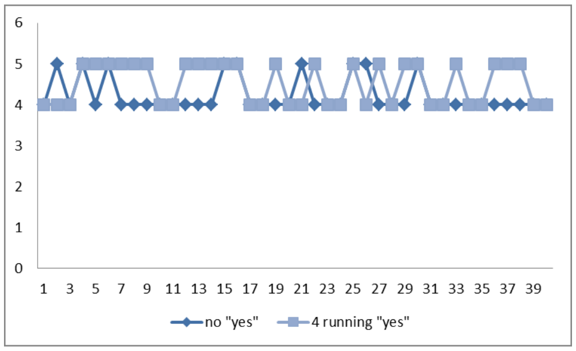

FIGURE 11: *Dequeue time comparison of Hybrid OS when no load and four "yes", y-axis unit: usec*

5 Conclusion

In this paper, we presented Hybrid OS that consists of Linux and a hardware sorter which replaces Red-black tree data structure used in Linux CFS scheduler. We implement Hybrid OS with Linux using FPGA and provide simulation and experimental results, and demonstrate the performance improvement and jitters reduction. In this approach, we only need to patch few lines of Linux code to use hardware sorter and no application code change is needed, which is different than other existing approaches. Hence, our approach could be used in wider practical application projects if commercial hardware product is available.

Implementing a hardware component for a generic data structure, such as a priority queue, could be easily accepted by chip manufactors than implementing entire operating system. Also a queue or priority queue is most commonly used data structure in operating systems. Hardware Tagged sorter, as a priority queue, could be used in other scheduler class, such as `SCHED_DEAD` or high resolution timer.

Though modern operating systems are rather complex and continuously evolving, few operating systems such as Linux is used by majority of industry. So identifying software components within popular operating systems and realize them by hardware accelerations would improve system performance and be accepted by industry.

References

[1] M. Sindhwani, T. F. Oliver, D. L. Maskell and T. Srikanthan, *RTOS Acceleration Techniques - Review and Challenges*, Real-Time Linux Workshop, 2004

[2] T. Klevin, *Get RealFast RTOS with Xilinx FPGAs,* Xcell Journal, Spring 2003

[3] L Lindh and F. Stanishewski, *FASTCHART Idea and Implementation,* IEEE Internaltional Conference on Computer Design (ICCD), Boston, 1991

[4] S. Nordstrom and L. Asplund *Configurable Hardware/Software Support for Single Processor Real-Time Kernels,* Internaltional Symposium on System-on-Chip, 2007

[5] S Moore and B. Graham, *Tagged Up/Down Sorter A Hardware Priority Queue,* The Computer Journal, Vol. 38, No. 9, 1995

[6] C. Wang, H. Wu, S. Huang, *A fast tagged sorter used in 100/10 MBPS routers,* International Conference on Consumer Electronics, 2000

[7] A. Morton, I. Song, J. Liu, *Hardware data structure for real-time scheduling and concurrency,* Proceedings of the IEEE International Conference on Field Programmable Logic and Applications, 2007

[8] S. Moon, J. Rexford, K. Shin, *Scalable hardware priority queue architectures for high-speed packet switches,* IEEE Transactions on Computers, Nov. 2000

[9] A. Morton and W. Loucks, *A Hardware/Software Kernel for System on Chip Designs,* ACM Symposium on Applied Computing, 2004

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

[11] A. Parisoto, A. Souza Jr., L. Carro, M. Pontremoli, C. Pereira, A. Suzim, *F-Timer: Dedicated FPGA to Real-Time System Design Support,* Proceedings., Ninth Euromicro Workshop on Real-Time Systems, 1997

[12] T. Nakano, A. Utma, M. Itabashi, A. Shiomi, M. Imai, *Hardware implementation of a real-time operating system,* The 12th TRON Project International Symposium, 1995

[13] P. Kohout, B. Ganesh, B. Jacob, *Hardware support for real-time operating systems,* international conference on Hardware/software codesign and system synthesis, 2003

[14] MicroC/OS-II, `http://micrium.com`

[15] Xilinx Inc., `http://www.xilinx.com`

[16] Mentor Graphics, `http://model.com`

[17] J. Corbet, *The high-resolution timer API,* LWN.net, `http://lwn.net/Articles/167897`

[18] D. Faggioli, F. Checconi, M. Trimarchi, C. Scordino, *An EDF scheduling class for the Linux kernel,*, Real-time Linux Workshop, 2009