Evaluating Task Scheduling for an FPGA Based Embedded Linux System

Minwoo "Jake" Lee

***Abstract*—Efficient application scheduling is critical for achieving high performance in embedded systems. This project builds the embedded Linux system on FPGA board, and running a small application, it evaluates the scheduling algorithms.**

***Index Terms*—Scheduling, Embedded Systems, FPGA, Embedded Linux.**

I. Introduction

As the complexity of embedded system grows, the needs for high performance increase. With limited resources, efficient scheduling is one of the key elements for achieving high performance. Many embedded systems also need real-time requirement or the performance at the level of real-time. For these systems, the efficient scheduling method is crucial.

Field Programmable Gate Array (FPGA) is an integrated circuit designed to be configured by users or designers. They can create their own custom logic in a short amount of time. FPGAs contain programmable logic blocks and a hierarchy of reconfigurable interconnects that allow the blocks to be wired together. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. Because it is flexible to design, and it can build up hardware fast, I chose FPGA board as my hardware platform.

An operating system, in a general sense, interconnects the software and hardware [1]. Operating systems give a software designer to develop variety of application without wrestling with hardware components. Embedded Linux is such an operating system running on small systems. By running embedded Linux, this project have many possible applications.

In embedded systems with limited resources, a CPU can be considered a resource that a scheduler can temporarily allocate a task. Thus using the CPU resource efficiently is very important, even with operating systems that possible can be slower than direct implementation. An important goal of a scheduler is to allocate CPU time slices efficiently while providing a responsive user experience.

The goal or this project involves the design and implementation of embedded Linux system that is running on FPGA board, that can run user application, and that can compare the performance when changing the scheduling algorithms. First thing to do for the project is understanding FPGA (Altera DE2) and generating a hardware description for next step. Second thing to do is running Linux on the configured board. Third step is running application–original goal was running an intrusion detection system (IDS). Because IDS requires more than 30MB of memory space, I could not run it on DE2 board. Fourth step is evaluating scheduling algorithms while running the application.

Because of the limitation in hardware and time constraints, I could not reach all the goals that I planned. However, I went through and achieved many of my goals. In the following section, I introduce the embedded Linux system and evaluation. Section II introduces the scheduling algorithms in Linux. Section III describes the designed embedded Linux system and the simple application. Section IV focuses on the comparison of scheduling algorithms through the test and observation. Section V concludes with a future works and comments. In Section VI, I add some comments about this projects.

II. Scheduling Algorithms

Linux offers three algorithms to deal with the task schedule [2]. One default scheduling algorithm is for normal processes, and the other two are for real-time tasks. Basically scheduling policy is based on ranking processes according to their priority. The process priority can be divided in static and dynamic priority. By default, the process priority is dynamic.

Static priority that never changes by the scheduler is assigned to real-time processes. In contrast, the scheduler keeps track of what processes are doing and adjusts their dynamic priorities periodically. In this way, processes that have been denied the use of the CPU for a long time interval are boosted by dynamically increasing their priority.

The first and default one is time-sharing technique. The CPU time is roughly divided into slices, one for each runnable process. If a currently running process is not terminated during the given time quantum, context switching may take place. Time-sharing is based on dynamic priority. Scheduler monitors the task activity, and if it is interactive process, it boost the priority. If it is a CPU hog, it will get a penalty [3]. Time-sharing relies on timer interrupts and is thus transparent to processes. No additional code needs to be inserted in the programs in order to ensure CPU time-sharing [4].

Second algorithm in Linux is First In First Out (FIFO) that uses the static priority. The static priority is always higher than the dynamic priority of the previous conventional tasks. When the scheduler assigns the CPU to the process, it leaves the process descriptor in its current position in the run-queue list. If no other higher-priority real-time process is runnable, the process will continue to use the CPU as long as it wishes, even if other real-time processes having the same priority are runnable.

The last algorithm is the algorithm for a Round Robin real-time process. When the scheduler assigns the CPU to the process, it puts the process descriptor at the end of the run-queue list. This policy ensures a fair assignment of CPU time to all real-time processes that have the same priority [4].

In addition to these three built-in algorithms, there was an effort to add one of the most popular real-time algorithm, Earliest Deadline First (EDF) by Dario Faggioli and Michael Trimarchi [5]. Since the previous algorithms cannot provide the guarantees a time-sensitive application may require and the latency is not deterministic and cannot be bound, it was hard to get a real-time performance. Figure 1 shows the the addition of EDF scheduler in Linux. Each task has its value for runtime and period that is equal to its deadline. At any time, the system schedules the ready task having earliest deadline. During the execution, the task's runtime value is decreased by a mount equal to the time executed.

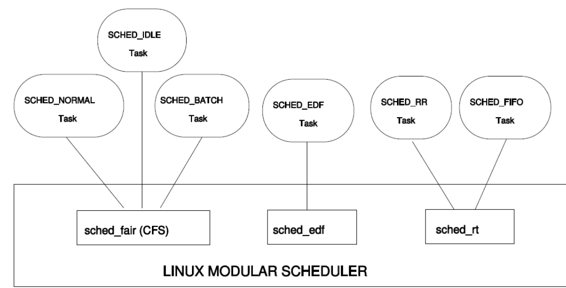

Fig. 1: Linux scheduler by Faggioli and Trimarchi [6]

When the runtime reaches to zero, the task is stopped until the system schedules it again.

III. Embedded System Design

Designing the target system starts from understanding the hardware. After learning the FPGA platform, I created the hardware description suited for my application. Third step is configuring and running operating system that mediates between application software and hardware resources. To add EDF algorithm for more test, it is necessary to change the kernel code in this step. After all these steps, the embedded system is ready. Any application can be compiled, built into the same image, and loaded on the system. I chose the simple JPEG application jpegview because of the limited resource–8MB of SDRAM memory.

A. Hardware

Figure 2 shows the Altera DE2 development board that I used to build the proposed system. DE2 board has Cyclone II 2C35 FPGA in a 672-pin package. All components such as LCD, switches, USB, RS-232, VGA, and etc. are connected to pins of the FPGA chip, allowing the user to control all the components. Below shows the brief specification from the manual.

FPGA: Cyclone II EP2C35F672C6 FPGA and EPCS16 serial configuration device

I/O Device: USB Blaster for FGPA configuration, Ethernet (DMA9000a), RS232, infra-red port, VGA, Audio, PS/2

Memory: 8MB SDRAM, 512KB SRAM, 4MB Flash

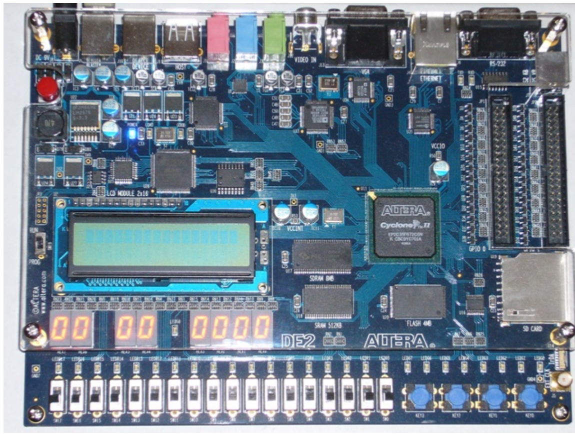

Fig. 2: Altera DE2 board

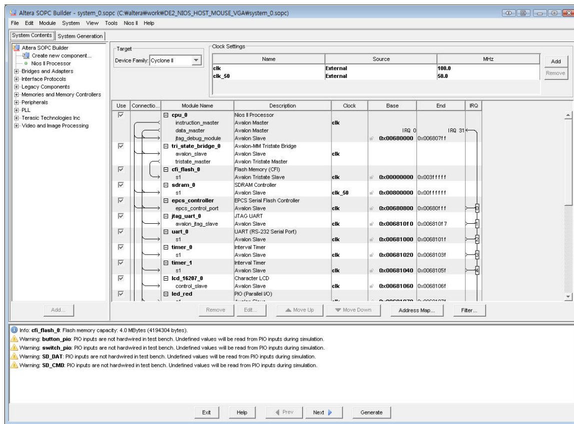

Fig. 3: SOPC Builder for board description

B. Hardware Description Generation

In order to be able to compile the Linux kernel and to run the hardware, it is necessary to do some work in Quartus and SOPC Builder first. Using SOPC Builder, I added flash device to generate the board description file. Figure 3 shows the hardware design of the target system.

C. μClinux

The operating system that I chose for the project is μClinux, one of the most popular embedded Linux. μClinux targeted platforms that lack memory management units (MMU). μClinux was first written for the Motorola DragoBall processor, and has since been ported to a large number of embedded architecture such as Altera Nios II, Xilinx Microblaze, ARM [7].

The lack of MMU imposes additional requirements. It do not provide support for shared libraries, so all the binaries must be statically compiled. This makes the size of the entire image grows, which is critical to this small embedded system.

To build the μClinux, binary tool chain (http://www.nioswiki.com/OperatingSystems/UClinux/BinaryToolchain) is required. After install and set-up, cross-compiler for Nios II processor is available. There are several documents that are useful to reference including the nioswiki (http://www.nioswiki.com/InstallNios2Linux), David Lariviere's technical report [7], and μClinux Tutorial Altera NIOS2 by Manfred Schmid **??**. Successfully ported μClinux has the boot loader, modified version of Linux 2.6.30 kernel, root file system with applications that I added. Figure 4 shows the block diagram.

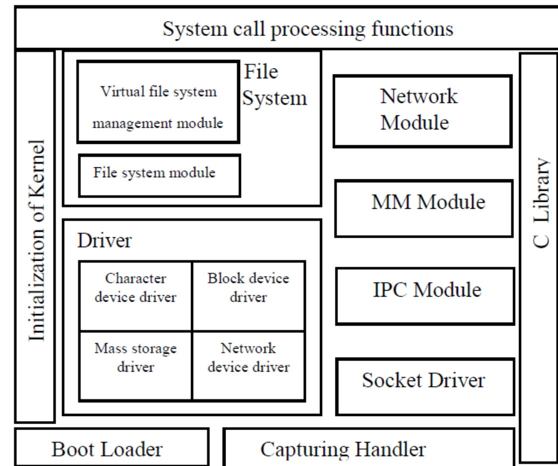

Fig. 4: uClinux Architecture [8]

D. EDF for μClinux and Issues

Faggiolli's work was great, and it speeded my project process much up greatly. Applying patches almost completed the work. There are some errors that I needed to fix, and additional codes to support Nios II were necessary. The modified kernel runs fine until I tested EDF scheduler. Modified kernel requires an application to specify the deadline and the worst case execution time (WCET). Listing 1 shows the code for EDF scheduling.

Listing 1: Modification of application for EDF scheduling

```
struct sched_param_ex sched_edfp;

int policy = SCHED_DEADLINE;
memset(&sched_edfp, 0, sizeof(sched_edfp));

sched_edfp.sched_priority = 13;
```

```
time_to_timespec(&sched_edfp.sched_deadline,
                10000); // Deadline
time_to_timespec(&sched_edfp.sched_runtime,
                1000); // WCET
int mask = 0x1 << 0;

sched_setaffinity(0, &mask);

if(sched_setscheduler_ex(0, policy,
  sizeof(sched_edfp), &sched_edfp)==0)
  printf("set process to real-time
        priority (EDF)\n");
else
  printf("Couldn't invoke real time
        scheduling priority [%s]\n",
        strerror(errno));
...
```

E. JPEG

Because of memory constraints, jpegview is the application that I chose. It is a very small and fast JEPG viewer program. When the program starts, it opens the given image files, allocates or initiates the JPEG decompression object and other parameters, decodes the images. This program uses libjpeg, and, of course, it should be compiled statistically. Now, the VGA driver handles to display the decompressed images on the screen.

IV. Test Results

Since the current version of libc and compiler does not support the function sched_setaffinity, I could not test with EDF. Given time constrains, instead of fixing this problem, I tested built-in scheduling algorithms only. For each scheduling algorithm, the same test were repeated three times. The values used on the plots are average of three tests.

Let's take a look at the graphs that records the performance over some period of time. Figure 6 shows the overall cpu utilization rate, Figure 7 shows the CPU utilization in user mode, and Figure 8 shows the number of interrupt requests (IRQ) over the time. There are not distinguished trend on Figure 6, but the range of variation is biggest in FIFO. TS, non-realtime scheduling method, oscillates less than other realtime scheduling. While

Figure 5 shows the excution time. Blue bar indicates the real time elapsed, red is the time spent in user mode, and the green bar represents the time in system mode. There are not huge difference in runtime since JPEG program is fast. However, FIFO spends the longest time while it spends the shortest in system mode. Round robin scheduler allow the process in user mode longest. While Time Sharing and Round Robin assign a lot in user mode process, FIFO assigns much less. FIFO maintains low utilization during the test time. TS and RR varies in their peak time since their time slice or quantum is different. IRQ plot shows another interesting trend. FIFO generates a lot of IRQs at first while RR generates very little. After a few cycle, IRQ rates for all three algorithms are gathered in the middle and does not change a lot.

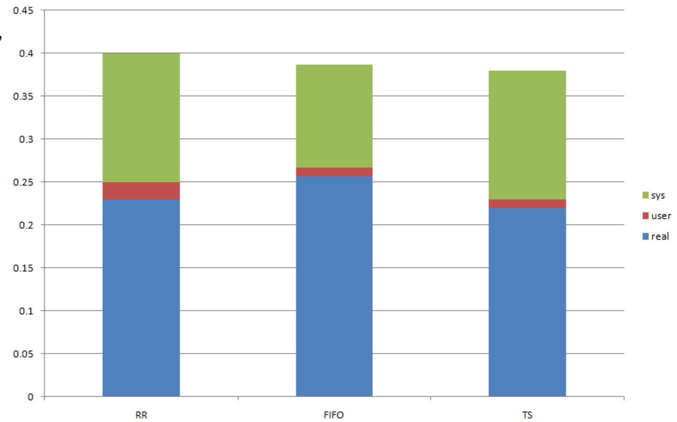

Fig. 5: Execution Time

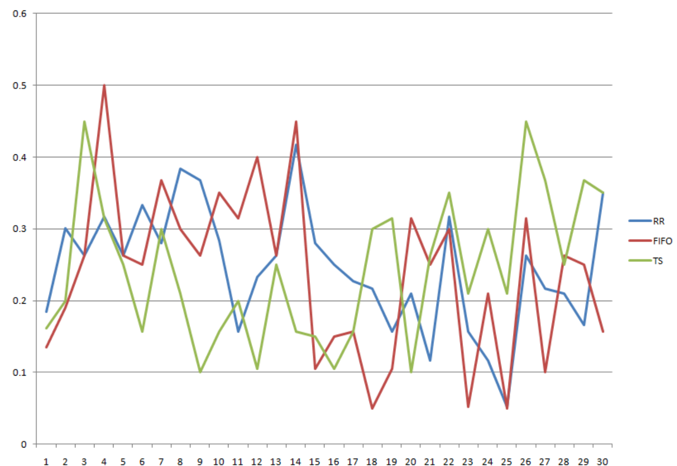

Fig. 6: CPU usage

V. Conclusion

JPEG program runs well on μClinux over Altera DE2 board with some memory limitation. Although fast and small JPEG comparison cannot show distinguishable difference between the three algorithms,

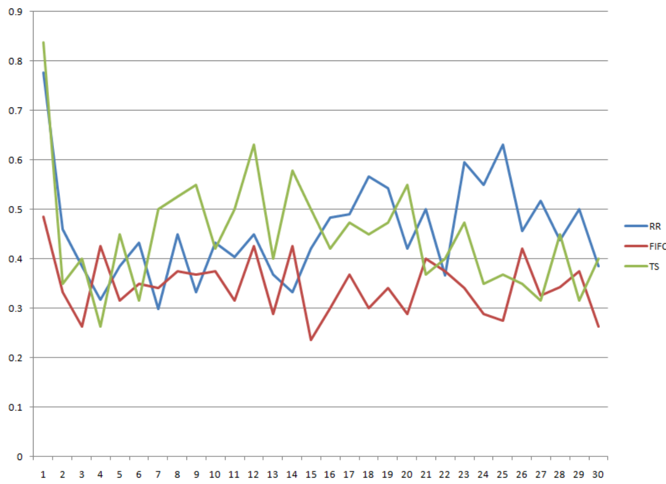

Fig. 7: CPU usage in user mode

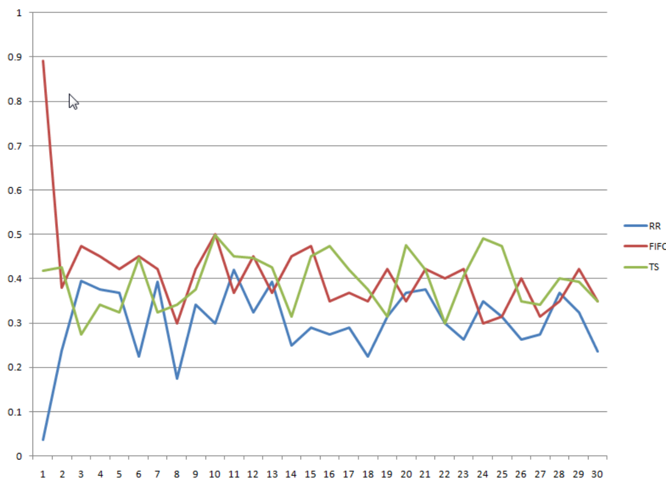

Fig. 8: Interrupt request

I could observed some trend. Since the number of tests, it is very hard to tell which algorithm works better for JPEG. Although I could not answer to the questions such as which algorithm works better for JPEG in DE2 or what the differences of them are, it was worthy experiment to attempts. I can extend my work to experiment with bigger applications such as MPEG or benchmark test. I might be able to find more obvious trends or differences.

In the future, I will test the EDF algorithm on the modified kernel. I also want to add Least Laxity First algorithm on the kernel. I might be able to study and import hybrid algorithm or adaptive algorithms on this system. If the memory problem is solved, I might be able to apply this system to bigger program such as security or artificial intelligent processing.

VI. Commnets

During this semester, I learned a lot about developing on Altera DE2 with Quartus tools, importing embedded Linux on the hardware platform, understanding kernel and Linux scheduler, and writing an application on embedded Linux. There was definite difference from developing in general purpose processor environment. I had to consider the memory limitation in the very early stage of modelling. Failure to measure my system requirements resulted in missing the scheduled milestones and finally changing my application. Overall, it was great experience designing and developing an embedded system from the scratch.

References

[1] M. T. Jones, “Inside the linux scheduler,” IBM Developer Works, 2006. [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-scheduler/

[2] “Process scheduling in linux,” http://www.cpccci.com/blog/2009/01/28/process-scheduling-in-linux/, January 2009.

[3] J. Aas, “Understanding the linux 2.6.8.1 cpu scheduler,” *SGI, 2005. http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf, accessed on August*, vol. 22, 2005. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.6385

[4] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel; 3rd ed.* Sebastopol, CA: O’Reilly, 2006.

[5] D. Faggioli, M. Trimarchi, and F. Checconi, “An implementation of the earliest deadline first algorithm in linux,” in *SAC ’09: Proceedings of the 2009 ACM symposium on Applied Computing*. New York, NY, USA: ACM, 2009, pp. 1984–1989.

[6] “An edf scheduling class for the linux kernel,” http://lwn.net/images/conf/rtlws11/papers/paper.16.html.

[7] D. Lariviere and S. A. Edwards, “uclinux on the altera de2,” 2008. [Online]. Available: http://hdl.handle.net/10022/AC:P:29611

[8] Z. Lu, X. Zhang, and C. Sun, “An embedded system with uclinux based on fpga,” *Pacific-Asia Workshop on Computational Intelligence and Industrial Application, IEEE*, vol. 2, pp. 691–694, 2008.