International Journal of Advances in Computer Science & Its Applications – IJCSIA
Volume 4: Issue 4 [ISSN 2250-3765]
Publication Date : 27 December,2014

Multi-Scheduling Technique for Real-time Systems on Embedded Multi-core Processors

[Abdulkadir Yaşar, Kayhan M. Imre]

Abstract*— Recent studies have shown that today's embedded systems require not only real-time ability but also general functionality for running non-real-time applications. In order to provide these two functionalities together on the same system, several techniques and frameworks have been proposed. Integrating different type operating systems on a multi-core processor is one of the most favourable approaches for system designers. However, this heterogeneous approach has some drawbacks. This paper introduces Multi-scheduling method for multi-core hardware platforms without running heterogeneous operating systems concurrently. In this technique, there are two schedulers in single operating system. One is for real-time applications and the other is for general or non-real-time applications. In Multi-scheduling approach, real-time and non-real-time applications run in the same operating system environment so the implementation and maintenance of the system become easier than heterogeneous approaches. We have implemented Multi-scheduling technique on Linux and benchmarked the interrupt latencies and stability of real-time applications. The results have shown that Multi-scheduling technique can be profitable to provide the real-time functionality for general purpose operating system.

***Keywords*—Multi-core, Multi-Scheduling, Embedded Systems, Real-time, Scheduling, Operating System, Symmetric-multiprocessing**

I. Introduction

In the last decade, the processor manufacturers place multiple processor cores in a single chip called System on Chip (SoC) to speed up the computation, to improve the performance and to reduce the cost. These processors may be composed of two or more independent cores based on symmetric or asymmetric multiprocessing architectures [1]. In addition to desktop or server PCs, multi-core processors are used in embedded systems for performance and economic reasons.

A typical embedded system is dedicated to perform functions such as real-time data control and digital signal processing. Unsurprisingly, embedded systems also require general non-real-time functionality as well as real-time (RT) functionality. Combination of these two functionalities is one of the most challenging problems for embedded and RT system developers. In order to overcome this problem, processor manufacturers usually produce heterogeneous multi-core processors [1, 9]. In heterogeneous processors, each core runs a different type of operating system (OS) to perform required functionality. For example, in a dual-core processor, a real-time operating system (RTOS) runs on one core, and a versatile or general purpose OS runs on the other. Memory and peripherals are isolated by hardware, or a low-level software called hypervisor. On the other hand, in homogenous processors, each core runs the same OS code, and share the main memory, peripherals and other resources.

Migration from single-core to multi-core processor brings a discussion about how to manage OS code over the cores in SoC. There are two suggested modes; asymmetric-multiprocessing (AMP) and symmetric-multiprocessing (SMP). In AMP mode, each core has its own copy of OS kernel code, and the codes are generally different from each other (heterogeneous OSes). On the contrary, in SMP mode, the same kernel code runs on each core synchronously (homogenous OSes). SMP OS dynamically balances the work between processor cores, and controls the resource sharing, e.g., main memory, between the cores [5, 12].

Despite the fact that the heterogeneous OSes in AMP mode are difficult and costly to maintain, they are widely used in embedded RT systems [1]. The reason behind this fact is that reserving the processor time for RT tasks in SMP mode is not trivial. In AMP mode, one or more cores run a RTOS in which RT application get the total control of the core(s) easily. The RTOS has a special scheduler, e.g., EDF scheduler, to meet strict timing constraints [11]. On the other hand, a general purpose OS has a scheduler, e.g., CFS, which gives tasks a fair share of a CPU's time.

It is essential that a RT system must respond actions or produce results within predefined timeframes [11]. In a RTOS, a task must gain immediate access to the processor to produce a timely response to an interrupt or action [9, 11, and 12]. Therefore, processor should be waiting in idle state for the most of its time. Although the processing speed is important for the quality of a RT system, it is not the primary purpose of an RT system. The primary purpose of an RTOS is to eliminate the surprises [12]. In other words; RTOS must provide a solid infrastructure to guarantee the response time of a task. However, in a general purpose OS, time for the completion of a task is unpredictable and may diverge.

In this work, we propose a new technique called Multi-scheduling for SMP multi-core embedded processors to enable to run RT tasks along with the general purpose non-real-time tasks. We have implemented our approach in Linux since it is the most widely used OS in embedded systems. In our approach, there are two schedulers running in a single OS environment. After booting on SMP system, one or more cores, selected in kernel configuration, change their scheduling policy to an appropriate RT scheduler. Therefore, RT tasks and general tasks are run on separate cores. We have measured the interrupt latencies and average task completion times of the multi-scheduling policy on a system containing an ARM Cortex-A9 dual-core processor. We have also carried out the same measurements for the standard kernel on the same hardware. Our results show that multi-scheduling

technique can be used to bring RT functionality to SMP homogenous multi-core processors.

The remainder of this paper is organized as follows: Section II reviews related works; Section III introduces the Multi-scheduling technique and its implementation on Linux; Section IV identifies the benchmark and comparison results; and finally conclusions and future works are drawn in Section V.

II. Related Work

Heterogeneous operating systems are widely used in embedded systems to integrate real-time and non-real-time functionality together. A low-level software called hypervisor is used to partition the hardware resources between OSes [4]. Moreover, physical partitioning techniques have been developed to run RTOS and general OS simultaneously on same system [1]. However, the physical partitioning techniques require hardware modifications on SoC to control the access lists to resources.

Linux is widely used OS in not only servers and desktops but also embedded systems. However, it suffers from lack of hard RT functionality. Although it is originally developed as general purpose OS, several RT infrastructures have been adopted to Linux kernel in recent years [5], [6]. The RT-patch provides several modifications such as low latency support and pre-emption into the standard Linux kernel to yield hard RT support [12]. Nowadays, the standard Linux also can be used in RT application but it provides soft RT infrastructure [4]. Researchers in [7] made first experimental analysis of RT performance of the standard Linux primitives on multi-core platforms.

In the recent years, numerous scheduling methods have been suggested for homogenous multi-core processors. Authors in [2] implemented a hybrid scheduling method to make the parallelism by partitioning an application into some parallel tasks. In [3] and [8], the authors implement a task splitting semi-partitioned scheduler for multi-core embedded systems. They show that semi-partitioned scheduling has better performance and low overhead than other partition-based scheduling methods. Moreover, authors in [14] have developed a loadable RT scheduler suite to support different scheduling algorithms on multi-core platforms. In [10], the researchers discuss an approach for supporting soft real-time periodic tasks in Linux running on high performance asymmetric multi-core platforms, or AMPs. In [15], a scheduling method is suggested for real-time systems implemented for multi-core platforms that encourage individual threads of multi-threaded real-time tasks to be scheduled together. The authors in [16] propose a hybrid approach for scheduling real-time tasks on large-scale multi-core platforms with hierarchical shared caches. In this approach, a multi-core platform is partitioned into clusters. Tasks are permanently assigned to the clusters, and scheduled within each cluster using the pre-emptive global EDF scheduling algorithm.

III. Materials & Methods

In this section, we will introduce what Multi-scheduling technique is and how it works. Then, we will mention about the implementation of Multi-scheduling technique in Linux.

A. *Multi-scheduling Technique*

Multi-scheduling is developed for SMP operating systems, where each core runs the same kernel code synchronously as if the system has a single-core processor. Most of the modern OSes support the SMP system. In SMP systems, one of the cores, generally CPU core-0 called primary-core is responsible for initialization of the hardware and all subsystems at boot time. After successful initialization, the same kernel code is copied to the other cores, called secondary-cores, on the SoC. Then, the tasks are assigned to cores to be run and load-balancing mechanism balances the work on the cores running the same scheduling policy.

OS kernel is composed of threads a.k.a. kernel threads such as interrupt handlers and kernel services. They are also handled by the system scheduler, running periodically depended on the CPU architecture and triggered by the CPU timer. In a multi-core platform, each CPU is triggered by its timer and runs the same scheduler code. For SMP systems, all kernel threads share the same context in the main memory. Therefore, additional synchronization codes, i.e., spinlocks, are used to provide consistency between multiple threads. All tasks stored in the memory are handled by schedulers in CPUs synchronically [1, 6].

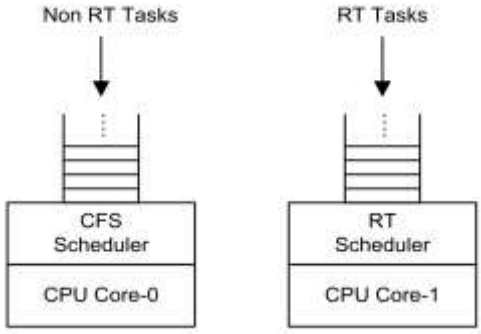

Figure 1. Illustration of Multi-scheduling

In Multi-scheduling technique, the shared context is copied and modified for one or more of the secondary-cores. We called these cores *rt-cores* where RT tasks will be handled on. Rt-cores will change its scheduling policy for RT task scheduling in their copied context. Moreover, load-balancing mechanism which balances the work between cores does not interfere with rt-cores; in other words, these cores are isolated from the other cores. For example, in Figure 1, CPU core-0 of a dual-core embedded system initializes and configures the hardware, and then, the core-1 runs a secondary startup code to initiate itself. In this secondary startup code for core-1, the scheduling policy of the second kernel image is changed to an RT scheduler and the load-balancing mechanism becomes aware of this.

B. *Implementation of Multi-scheduling*

Multi-scheduling technique is developed for Linux. It is initialized and started to run in the boot process. The boot process of embedded systems is different from desktop or server PCs. When the power button is pressed, a small boot-loader software finds the OS image and loads it to the main memory and then OS initialization process begins. In the Linux SMP environment, core-0 is primary-core for initialization. When the initialization is completed, primary-core will signal the secondary cores to boot a specific kernel code called ***secondary_start_kernel()***. When the secondary cores boot the same Linux image, they will enter Linux at a specific location so they simply initialize the resources (e.g. MMU and caches) specific to their cores only. The secondary cores don't reinitialize resources that have already been configured, and they just execute the idle process with PID 0. Consequently, each core on the system has its own environment containing a scheduling policy triggered by a timer specific to the core.

We have carried out some modifications to the Linux operating system in both user-space and kernel-space. In kernel-space, ***secondary_start_kernel()*** code has been modified to run a different scheduling policy for RT functionality. First of all, rt-cores are selected in kernel configuration for multi-scheduling and the selected core list is stored to allow tasks to run on them later. In ***secondary_start_kernel()***, the shared context is copied for rt-cores and the rt-cores re-initialize the scheduling mechanism. Each rt-core changes its scheduling policy to SCHED_RR or SCHED_FAIR policies, defined in the Linux Kernel for RT applications, rather than SCHED_OTHER, aka CFS (Completely Fair Scheduling) default policy in Linux.

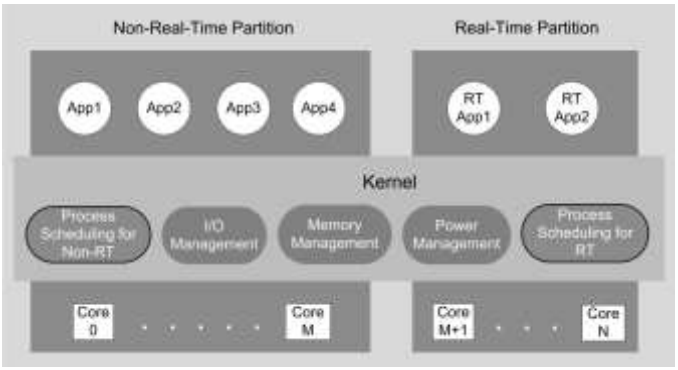

Figure 2. Detailed view of the Multi-scheduling enabled Operating System Environment

In Linux, each core has its own task queue (*runqueue*) for keeping the task to be run on that core. The other modifications in kernel-space have been applied to Linux *load-balancing* mechanism working on the runqueues to balance the tasks between CPU cores on the system. The load balancing mechanism for rt-cores does not interfere with the corresponding mechanism for the cores reserved for non-real-time tasks. Consequently, the whole environment is partitioned into two separate environments; RT and non-RT. The tasks are also split into two groups, and each task runs on a corresponding core depending on its type whether it is RT or non-RT. This is depicted in Figure 2.

Apart from these modifications in kernel-space, some additions more are needed in user-space for Multi-scheduling. We have implemented a *task-assigner* utility to assign tasks either to RT or non-RT partition. The utility runs a task first and then changes its scheduling policy to SCHED_RR if the task is for RT partition; otherwise the policy is set to SCHED_FIFO for non-RT tasks. The RT tasks are assigned to rt-cores, and the rest assigned to the other cores. In addition to forking the task structure and changing the task's scheduling policy, the *task-assigner* updates the allowed core(s) list for the task. Therefore, the tasks assigned to RT partition are scheduled by the RT scheduler in the kernel space.

IV. Real-time Performance Results and Discussions

We have tested the RT performance of Multi-scheduling technique on the pandaboard widely used in Embedded community as reference design. It has a dual-core ARM cortex-A9 powered by TI's OMAP4460 microprocessor. Therefore one of the CPUs runs RT tasks and the other is for IT tasks and general OS operations. We have prepared two patches and applied the Linux Kernel version 3.4. One of the patches called *Msched-P1* runs RT tasks with SCHED_RR scheduling policy and the other called *Msched-P2* runs with general Linux scheduling policy SCHED_OTHER.

Thanks to the Linux community, there are many RT performance and benchmark test tools. For example, *cyclictest* is one the of most known RT. It measures the amount of time that passes between when a timer expires and when the thread which set the timer actually runs [12]. This value is the latency for that timer wakeup. As mentioned before, the interrupt latencies should be minimum and all response durations must be close as much as possible; in other words a RT system must produce stable results for same work in any case. We used *cyclictest* to measure interrupt latencies and *gpio-toggle* test to estimate the stability.

At first, we have measured the interrupt latency of the Standart Linux over CPU workload. In this test, we load work increasingly to CPUs by using *stress* program widely used in CPU tests. In Table 1, the more CPU stress increases, the more the latency to interrupt lenghtens. This is because, the standart Linux OS scheduler can not allocate enough time for timer interrupts. It tries to share the resources to tasks fairly and do not care about the RT tasks.

TABLE I. AVERAGE INTERRUPT LATENCY OF STANDART LINUX

Avg. Interrupt Latency of the Standart Linux over CPU work						
CPU workload	% 0	%20	%40	%60	%80	%100
Avg. Latency (us)	84	3382	6570	10370	14706	17780

In Figure 3, cyclictest results are shown for Msched-P1 and Msched-P2 patches. As you can see, although the CPU workload increases, the average interrupt latency do not change so much because the RT tasks run on a different CPU and are not affected by non-RT tasks' workload on the system. This graph also says that SCHED_RR scheduling policy used in Msched-P1 has better performance than SCHED_OTHER used in Msched-P2. Moreover, in CPU idle, Multi-scheduling patches nearly two times better performance than the standart Linux.

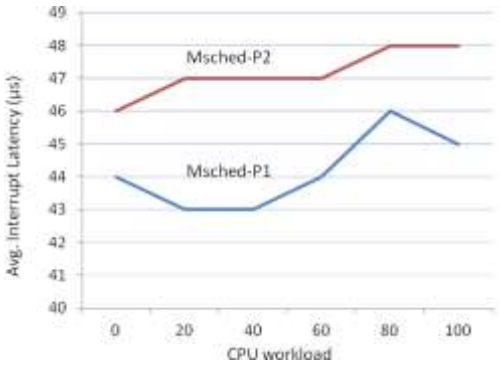

Figure 3. Interrupt Latency results of Multi-scheduling patches over CPU workload

Secondly, *gpio-toggle* test toggles a GPIO pin on the board and estimates the duration. In Figure 4, the duration of toggling in the standart Linux is not stable. Time for the

Figure 4. Scope output of GPIO toggling in the standart Linux

toggling changes in some cases as you can see on the falling edge of the output signal. On the other hand, in Figure 5, the duration for GPIO toggling in Multi-scheduling technique is more stable and shorter about two times than the standard Linux. As we mentioned before, the stabilization of processing a task in any case is more important for RT systems. This toggling test shows that the processing times of RT tasks is stable in Multi-scheduling technique.

Figure 5. Scope output of GPIO toggling in Multi-scheduling enabled Linux

v. Conclusion

In this paper, we proposed a new approach Multi-scheduling to run RT and non-RT tasks in a single operating system. It is based on the partition of the cores in the multi-core processor into two groups, and two different environments are created for RT and non-RT tasks.

Multi-scheduling technique only separates the cores on the system not the other resources such as main memory, USBs, GPIOs and other controllers. In order to provide a better RT and non-RT environment partition in the single OS, all resources on the system must be separated. For example, if a RT task and a non-RT task want to use the same resource, e.g., USB0, on the same time, it will reduce the overall performance and may cause the deadlocks. The current Multi-scheduling technique does not separate the other resources. In this work, we just want to show that creating two different environments may provide both RT and IT functionalities in the single OS.

The interrupt latency and stability results have shown that Multi-scheduling technique can be a good approach to provide RT functionality for general OSes without using heterogeneous OSes. On the contrary of heterogeneous approach, a single OS environment is used for all tasks. This provides two main advantages to system developers. One of them is Inter-process communication between RT and non-RT tasks. The other and more important advantage is about the system development and maintenance. In heterogeneous approach, system developers configure two different OS; a general OS and a RTOS. A Failure in one of the heterogeneous OSes causes the whole system come down. Moreover, developers spend more time to learn different OS environments. This may increases the costs for production.

Multi-scheduling is a valuable technique to provide RT functionality for general purpose operating systems. It may be considered as one of the most major approaches for Real-time systems in multi-core embedded systems. For future work, we want to extend Multi-scheduling technique to cover all resources in the system. Moreover, we will provide tools to control the Multi-scheduling from user-space easily.

References

[1] T. Nojiri, Y. Kondo, N. Irie, M. Ito, H. Sasaki, H. Maejima, "Domain Partitioning Technology For Embedded Multicore Processors" Published by the IEEE Computer Society, pp. 7-17, November 2009.

[2] P. Tan, "Task Scheduling of Real-time Systems on Multi-Core Architectures" in Second International Symposium on Electronic Commerce and Security (ISECS), Nanchang, China, May 2009.

[3] Y. Zhang, N. Guan, Y. Xiao, W. Yi, "Implementation and Empirical Comparison of Partitioning-based Multi-core Scheduling" in Industrial Embedded Systems (SIES), Sweden, June 15-17, 2011.

[4] S. Kai, Y. Liping, "Improvement of Real-time Performance of Linux 2.6 Kernel for Embedded Application", Proc. of International Forum on Computer Science-Technology and Applications (IFCSTA), December 25-27, 2009.

[5] N. Vun, H. F. Hor, J. W. Chao, "Real-time Enhancements for Embedded Linux", published in 14th IEEE International Conference on Parallel and Distributed Systems (ICPADS), Melbourne, Australia, December 8-10, 2008.

[6] C. Bi, Y. Liu, R. Wang, "Research of Key Technologies for Embedded Linux Based on ARM", in International Conference on Computer Application and System Modeling (ICCASM 2010), Taiyuan, Chine, October 22-24, 2010.

[7] Y. Zhang, C. Gill and C. Lu, "Real-Time Performance and Middleware for Multiprocessor and Multicore Linux Platforms" in 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Beijing, China, pp. 437-446, August 24-26, 2009.

[8] M. Shekhar, A. Sarkar, H. Ramaprasad, F. Mueller, "Semi-Partitioned Hard-Real-Time Scheduling Under Locked Cache Migration in Multicore Systems" in 24th Euromicro Conference on Real-Time Systems, Pisa, Italy, pp. 331-340, July 11-13, 2012.

[9] H. Tomiyama, S. Honda, H. Takada, "Real-Time Operating Systems for Multicore Embedded Systems" in International SoC Design Conference (ISOOC), 2008.

[10] J. M. Calandrino, D. Baumberger, T. Li, S. Hahn, and J. H. Anderson, "Soft real-time scheduling on performance asymmetric multicore platforms," Proc. of the 13th IEEE Real Time and Embedded Technology and Applications Symposium(RTAS'07), Bellevue, WA, USA, April 3-6, 2007.

[11] A. Mohammadi and S. G. Akl, "Scheduling Algorithms for Real-Time Systems" unpublished, supported by the Natural Sciences and Engineering Research Council of Canada, 2005.

[12] S. Rostedt, D. V. Hart, "Internals of RT-patch" in Linux symposium, Ottawa, Canada, July 18-20, 2007.

[13] R. Love, Linux Kernel Development 3rd Edition, published by Addison-Wesley Professional, 2010.

[14] S. Kato, R. Rajkumar, and Y. Ishikawa, "A Loadable Real-Time Scheduler Suite for Multicore Platforms," Technical Report CMUECE-TR09-12, 2009

[15] J. H. Anderson and J. M. Calandrino, "Parallel real-time task scheduling on multicore platforms," Proc. of The 27th IEEE Real-Time Systems Symposium (RTSS'06), Rio de Janeiro, Brazil, December 5-8, 2006.

[16] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger, "A hybrid real-time scheduling approach for large-scale multicore platforms," Proc. of the 19th Euromicro Conference on Real-Time Systems (ECRTS'07), Pisa, Italy, July 4-6, 2007.

About Author (s):

Abdulkadir Yaşar received the B.S. degree and currently studying the M.S. degree in Computer Science and Engineering from Hacettepe University, Ankara, Turkey, in 2011. He has been working as Embedded and Real-time Design Engineer in Aselsan Inc. since 2011.
His current research interests include the hardware/software codesign and Real-time system design for multiprocessor system-on-chips.

Kayhan M. Imre received his B.Sc., M.Sc., degrees in Computer Engineering from Hacettepe University, Ankara, Turkey and his Ph.D. degree in Computer Science from University of Edinburgh, Scotland, in 1985, 1987 and 1993 respectively. He is currently an Assistant Professor at the Computer Engineering Department, Hacettepe University.
His research interests are in parallel processing, parallel and distributed simulation and real-time systems.