# An Efficient Partitioned Scheduling Mechanism for Real Time Task Execution in a Multicore System

Lydia M
*Student, Computer Science Engineering and Department,*
*Rajalakshmi Engineering College, Tamil Nadu, India*

Benedict Jayaprakash Nicholas
*Assistant. Professor,Computer Science and Engineering Department,*
*Rajalakshmi Engineering College, Tamil Nadu, India*

**Abstract- Every task in an operating system is set with a priority. A non real-time task executing in the CPU is interrupted when a real-time task requests the CPU for execution. This is because the real-time tasks are always given higher priority than the non real-time tasks. Similarly the real-time tasks are in turn interrupted by hardware and software interrupts. In order to reduce this execution overhead of tasks we need to minimize the waiting time and the turnaround time of the task's execution. To achieve this in the proposed work the CPU cores are partitioned among the Linux tasks and the real-time tasks. A CPU with four logical cores is split in such a way that the CPU core 0 is allotted for Linux tasks, CPU core 1 and core 2 are allotted to be used as worker cores to execute real-time tasks and the CPU core 3 is allotted to be used as a scheduler core or the master core. A scheduler core is a core where all the scheduling policies for the tasks in worker cores are implemented. Further the proposed method also separates the worker cores and the master core from the kernel load balancing and scheduling. By separating the cores from default scheduling, the real time scheduling such as Partitioned Earliest Deadline First scheduling can be applied specifically to these cores. Additionally all the software and hardware interrupts are redirected to the Linux core. Thus the proposed system reduces the execution overhead of real-time tasks, by parallel handling of real-time and non real-time tasks.**

**Keywords – Complete Fair Scheduler (CFS), Partitioned Earliest Deadline First scheduling (PEDF), real-time tasks, non-real-time tasks, multicore system, CPU pinning.**

## I. INTRODUCTION

Multicore systems are systems in which multiple cores are embedded in a single processor chip. There are two types of multicore systems that are classified based on their design. The types include symmetric multicore system and asymmetric multicore system. To detail these types of multicore systems the best statement could be, a symmetric multicore is one where all the cores are identical with same design, capacity and is embedded within the same processor chip. The cores in symmetric multicore system can be used to execute different tasks in a similar way. Conversely in case of asymmetric multicore system, of the multiple cores some cores have a totally different design than others. Even the capability of each core in the multicore system varies widely from each other. However here we make use of the symmetric architecture that uses multithreading concepts to handle the task assignment to each core. In order to design our own scheduler we make use of Intel quad core processor which has a build of symmetric multicore architecture. Intel quad core has 4 logical cores of CPU that are of same build and capabilities. These logical CPU cores helps to improve the parallel execution of multiple tasks thus reducing the waiting time for a task's execution to complete. Thus the multicore platform enables the logical CPUS to communicate with other for the assignment and perfect sharing of tasks between them. Multicore architecture is thus an advancement in the field of parallel computing. Additionally it reduces the amount of power consumed and heat dissipated by the system that uses this architecture of multicore.

According to the way the system should respond to an input arrived, the job to be executed can be classified into real-time and non real-time job. A real-time job is given a diverged time interval to complete its execution. A real-time job thus always has an order to meet the specified deadline. In case of deadline miss the whole task comprising these real-time jobs may stop functioning in the right way. The best example of a real-time task could be online transaction. It should complete in an order as specified within a given deadline, failing which may lead to catastrophic results. Conversely on the other hand a non real-time task is one that does not have any deadline for completion. It may be completed at any time wished. The real-time tasks may be further classified into hard and soft real-time tasks. Hard real-time tasks match the explanation given above for the real-time task. It should strictly complete within the predicted deadline. The soft real-time tasks have some relaxation. It also has an allotted deadline for completion but a deadline miss is acceptable and does not lead to catastrophic results. However the deviations to deadline are accepted the deadlines are mostly met by the soft real-time tasks.

Since real-time jobs require a high processing capacity to meet its deadline, multicore systems work well with the real-time tasks. Multicore system enables parallel processing that reduces the interruptions caused by frequent preemptions of low level tasks thus enabling the tasks to complete its processing with high speed, neglecting the preemptions to a reasonable extent. In the multicore system each logical CPU core has its own level 1 cache, registers and execution resources. With all these resources owned independently each core can act independently executing the tasks assigned to it. The one unit that manages the distribution of jobs across the available CPU cores is known as scheduler.

There are different types of multicore schedulers in use. The one most popular Linux scheduler is the Completely Fair Scheduler (CFS). CFS was an enhancement to O (1) scheduler. The O(1) scheduler had run queue where all the tasks ready for execution was made to wait and it had a priority list where the ready to execute tasks were assigned priorities based on their importance to the system. The execution followed this priority, executing the highest set of priority tasks first, in concurrency. Complete Fair Share scheduler followed O(1) scheduler. It uses a data structure of red-black tree. Equality prevails in the treatment of processes rather than prioritizing. The tasks are placed in the nodes of red-black tree according to the allotted time for which they can remain in the processor before they are context switched. The processes aligned at the leftmost node of the tree is said to utilize the CPU at the time it is positioned in that node. It is an improvement in the scheduling complexity to O (log n). The most popular scheduler in windows operating system is Solaris scheduler.

## II. SYSTEM MODEL

There are various scheduling policies used with the above mentioned scheduler classes. Some commonly used scheduler policies are SCHED_FIFO, SCHED_RR, SCHED_IDLE, SCHED_BATCH etc. SCHED_FIFO is also known as first in first out scheduling policy. It is a non-preemptive policy. The process stays in the CPU as long as it needs the CPU. The processes are allotted the CPU based on the priorities set under a condition. SCHED_RR is also known as the round robin scheduling policy. Here a quantum time is assigned to the CPU so that every process can access the CPU only for that time quantum before they are preempted and the context is switched to the next higher priority process. When all the processes have executed once in the CPU the sequence begins again from the first process until all the process completes its execution. SCHED_BATCH executes the processes in batch style using round robin scheduling policy. SCHED_IDLE runs at the background all the tasks with lower priority.

Optimal static-priority scheduling or optimal dynamic-priority scheduling can be used to set priorities to the processes. These priorities decide which task should execute first based on some predetermined conditions. Rate Monotonic Scheduling (RMS) is an optimal static-priority scheduling where the priority is set based on the period of the process. The process with the shortest period is assigned the highest priority and the process with a long period of execution is assigned the lowest priority. The highest priority process is executed first in the CPU. On the other hand Earliest Deadline First Scheduling (EDF) is an optimal dynamic-priority scheduling where the priority is set based on the deadline of the process. The process with nearest deadline is assigned the highest priority and the process with far deadline is given the lowest priority. The highest priority process is executed first in the CPU.

In order to minimize the execution overhead of real-time processes the available logical CPU cores are partitioned. We use Intel i5 processor which is a quad core processor to implement this system model. A quad core processor has four logical CPU cores named as core0, core1, core2 and core3. We perform CPU shielding to reserve cores for specific tasks. CPU core 0 is reserved for executing all non real-time and soft real-time tasks of Linux operating system. Core 1 and core 2 of the CPU are dedicated to serve all real-time tasks specifically hard real-time tasks. The CPU core 3 is allotted to carry out all the scheduling functions specific to the real-time tasks hence the name scheduler core. Since it is in the scheduler core all the scheduling decisions are taken it is known as master core. Since the decisions taken by the master core are carried out by the cores allotted for real-time tasks, the real-time cores are known as worker cores. The scheduler core has a global release queue that queues all the tasks that has completed one round of its execution. The queued jobs are scheduled by the real-time scheduler, Partitioned Earliest Deadline First Scheduler (PEDF). The jobs when ready for execution are moved to the appropriate ready queues in the worker core. The jobs in the ready queue are allotted to the worker cores that are free at that moment dynamically. The worker cores and the master core are isolated from the default Linux scheduler (CFS). Hence PEDF scheduling handles the worker cores and the master core while the Linux core is handled by the default scheduler CFS. Thus proposed system model enables the real-time and non real-time tasks to execute in parallel in their respective cores eliminating the higher priority tasks from interrupting the execution of lower priority non real-time tasks thereby increasing the execution time of the non-real time tasks. This model hence eliminates the execution overhead of real-time tasks significantly.

## III. RELATED WORK

Steve Brosky and Steve Rotolo in [2] implemented a shielded processor technique where a processor is reserved for a particular jobin a multiprocessor system. This was an enhancement to the use of preemption patches which delivered the system from interrupt overhead to some extent. But the preemption patches worked only in the absence of graphics user interface and networking connectivity to the network. Shielded processor technique introduced in [2] have overcome this limitation and worked in the presence of graphics display and networking. Brosky in [3] introduces the concept of shielded CPU. [3] Tailors shielding the entire processor in a multiprocessor system to an individual CPU core that can be reserved to a high priority real-time tasks. Since a real-time task always preempts the non real-time tasks execution, executing these high priority real-time tasks in a separate dedicated logical CPU core enhances the task execution. This also enables the hard real-time tasks to complete its execution without a deadline miss, as the result after the deadline is considered to be incorrect.

LITMUS$^{RT}$ [4] is a testbed for empirically comparing real-time multiprocessor schedulers. As we have discussed above the various schedulers for multicore system such as first in first out scheduler, round robin scheduler, earliest deadline first scheduler and rate monotonic scheduler can all be compared with one and other for efficiency and performance on a set up environment in an operating system such as Linux. The schedulers can be activated on the jobs running in the operating system and finally comparisons can be made to judge the best scheduler of all for a specific environment. This testbed can also be used to design our own scheduler for scheduling the tasks in the multicore environment. LITMUS$^{RT}$ is the first attempt to compare partitioned scheduling and global scheduling on their schedulability in a multicore real-time environment that supports both hard and soft real-time tasks.

Having used the terms partitioned and global scheduling a question may arise on their description. In the global scheduling approach [5] the tasks ready for execution executes in any available CPU cores. No CPU core is reserved for any task. Global scheduling supports task migration from one core to another. All the available logical CPU cores share a common ready queue. Unlike the global scheduling approach in the partitioned scheduling approach the cores are partitioned. In other words some specific logical CPU cores are reserved for the execution of defined tasks. The tasks with reserved CPU cores may be real-time tasks. Partitioned scheduling does not allow tasks to migrate between the reserved cores. There is another approach of scheduling known as clustered scheduling [8]. In the clustered scheduling the logical CPU cores are grouped into clusters. The tasks here are allowed to migrate between the cores that belong to the same cluster. It does not permit the tasks to migrate between the clusters. One of the scheduler that was designed to operate as a global scheduler is Global Earliest Deadline First Scheduling short called as GEDF scheduler. EDF was also designed to operate as a partitioned scheduler and as a clustered scheduler called as Partitioned Earliest Deadline First Scheduler (PEDF) and Clustered Earliest Deadline First Scheduler (CEDF) respectively. In our work we make use of Partitioned Earliest Deadline First scheduler to schedule the tasks in the multicore system. A. Bastoni, B. B. Bradenburg and J. H. Anderson in [8] have made a comparison between global, partitioned and clustered multiprocessor EDF schedulers.

In [11] A. Bastoni, B. B. Bradenburg and J. H. Anderson have introduced a new scheduling approach known as semi-partitioned scheduling. Semi-partitioned scheduling is similar to the partitioned scheduling approach but unlike the partitioned scheduling its allows specifically few tasks to migrate between reserved cores. Additionally it also permits the cores to share the execution of a specific task by splitting the task to execute in different cores and then combining the results.

Exsched [12] is an external scheduler framework that provides us with a platform to develop our own scheduler without making any modification to the base operating system through an interface. It allows us to execute a scheduling policy as an external plug-in. Thus Exsched enables us to apply our own scheduling policies to specific cores without changing the system configuration.

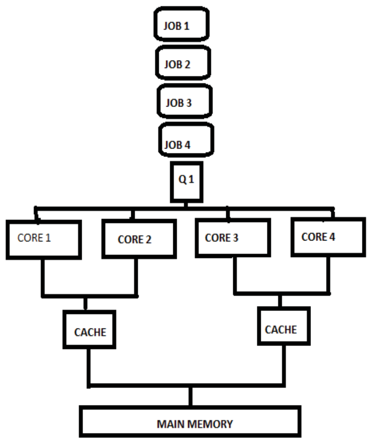

Fig. 1 Global scheduling

A scheduler class known as SCHED_DEADLINE was introduced in [6]. This scheduler class was implemented by making use of Earliest Deadline First scheduling policy. It is also known as SCHED_NORMAL. SCHED_FIFO and SCHED_RR take precedence over this class.Real Time Application Interface (RTAI) is an extension to real-time applications. Generic purpose Linux operating system supports only soft real-time and non real-time tasks. It does not support hard real-time job executions. In order to enable the operating system to support hard real-time tasks that has strict orders to meet the predetermined deadlines RTAI extension was introduced. This makes Linux kernel to coexist along with the Real Time kernel, thus providing a platform for both non real-time tasks and real-time tasks including both hard and soft real-time tasks.

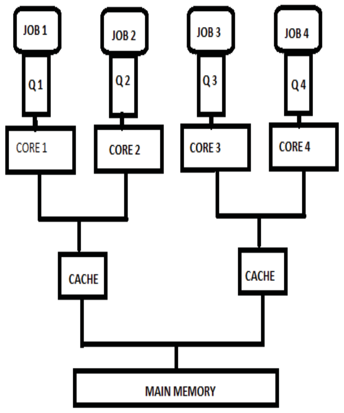

Fig. 2 Partitioned scheduling

## IV. PROPOSED METHODOLOGY

The proposed system is implemented on a Linux 64-bit operating system that runs on top of an Intel core i5-3337U CPU processor with an installed memory of 4GB RAM. The Linux kernel 4.1.3 has been patched up with a LITMUS$^{RT}$ plug-in. The commercial Linux kernel does not support real-time task execution. To provide a real-time extension to the general purpose Linux, the Linux kernel is patched up with the LITMUS$^{RT}$ plug-in. It provides the development stage to design our own scheduler that performs task allocation to the available free logical CPU cores. Figure 4 shows plug-in of LITMUS$^{RT}$.

The available CPU cores are partitioned into three sets. The first set is called Linux core. It consists of a single logical CPU core and it is responsible for running all the non real-time and soft real-times tasks of the Linux OS. All the hardware interrupts are redirected to the Linux core. The second set of cores is known as worker core. The worker cores are responsible for all the hard real-time tasks that has deadline constraints.

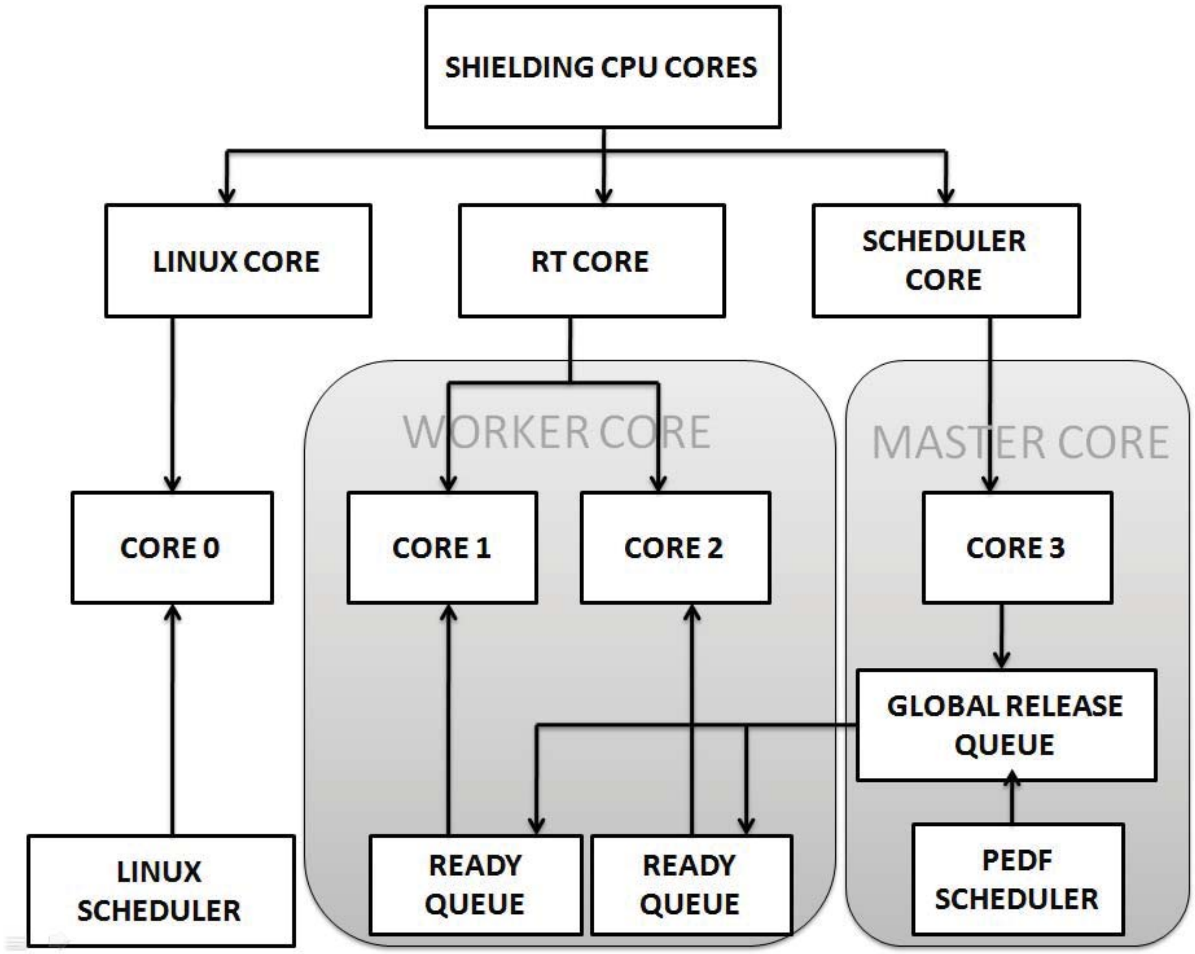

Fig. 3 System model

The available CPU cores are partitioned into three sets. The first set is called Linux core. It consists of a single logical CPU core and it is responsible for running all the non real-time and soft real-times tasks of the Linux OS. All the hardware interrupts are redirected to the Linux core. The second set of cores is known as worker core. The worker cores are responsible for running all the hard real-time tasks that has deadline constraints. It consists of two logical CPU cores. In other words two logical CPU cores are reserved for exclusively executing the real-time tasks. The third and the final set of core is known as master core. It consists of a single logical core. It is responsible to schedule the real-time tasks that have to be executed in the worker cores. The cores are reserved for the above mentioned functionalities by using the taskset command. The syntax of the command is as follows: taskset –cp core-no process-id. All the real-time process id are given with the core no core 1-2. So the current affinity list of that particular process is set to the CPU affinity 1-2 i.e. the process can execute in either core 1 or core 2. Similarly all the non real-time tasks are set with the affinity core 0. Furthermore all the interrupts of the hardware are set with an irq affinity to reserve CPU core 0 for handle all the interrupts. The syntax for changing the irq affinity is as follows: echo core-no-hexa > /proc/irq/interrupt-no/smp-affinity.

In order to use Partitioned Earliest Deadline First scheduler on the worker cores we need to isolate the worker and the master cores from the default Linux scheduler CFS i.e. Complete Fair Scheduler. This is achieved with the help of the command isolcpus. The syntax for isolcpu is as follows: isolcpus=<core-number 1, core-number 2,core-number n>. Once the core is isolated we can change the scheduling policy from Linux scheduler to Earliest Deadline First Scheduler.

A global release queue is constructed to be used with the master core. The master queue handles the ordering of jobs in the global release queue. The global release queue holds the jobs yet to be executed. Jobs in the global release queue are place according to its priority the jobs with the highest priority that is the jobs with the nearest deadline is placed first in the queue to its left. The jobs with decreasing priorities, in other words the jobs with farther deadlines are placed towards the right of the global release queue. The jobs in the queue construct are scheduled using Partitioned Earliest Deadline First scheduler by the master core.

A ready queue is constructed one per worker core. The ready queues hold all the jobs that are ready for execution and are waiting for the CPU core to be allotted. Whenever the timer triggers the execution of a job placed in the global release queue, the runnable job is moved to the ready queue from where it will be assigned to the working core attached to that particular ready queue. These ready queues are managed by the worker cores allotted to it. In the existing related works the CPU affinity is assigned statically that is it is told in prior that a particular real-time task should execute only in a particular worker core. In the proposed methodology we try to assign the affinity dynamically. When the jobs in the master core are ready to execute, it will be moved to the ready queue of a worker core that is currently free. The worker cores are not reserved between the real-time jobs.

## V. RESULT

Experimental studies were made by setting the scheduler policy to PEDF scheduler and CFS scheduler. Varying num The study proved that PEDF scheduler yields better performance when compared to CFS scheduler in this partitioned framework. The amount of time given to a job to execute in the CPU is more in partitioned scheduler. The execution time is less and the CPU time is more for the jobs in partitioned scheduling technique.

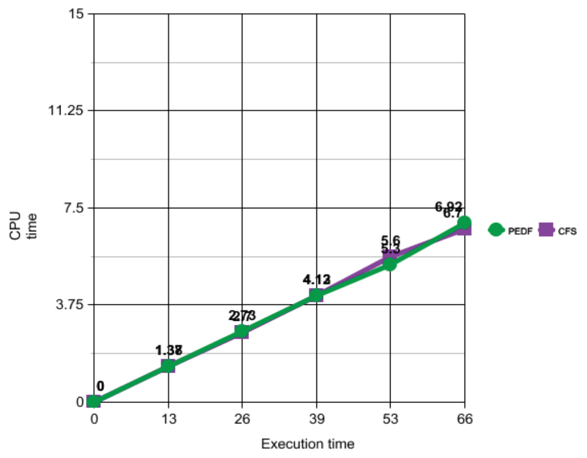

Fig. 4 CPU execution time of a task in an unpinned processor framework

On the other hand the execution time is more and the CPU time is less when using CFS scheduler. Hence PEDF scheduler is found to dominate CFS scheduler in performance on the framework with processors pinned. This is shown graphically in figure 4 and 5. The CPU time is plotted for the given set of actual execution time. As the execution time increases CPU time for the jobs also increases. The increase in the CPU utilization time with respect to the execution time is greater for PEDF scheduler when compared to the Linux scheduler (CFS). Therefore performance of partitioned scheduler is evaluated to be better when compared to the Linux default scheduler CFS.

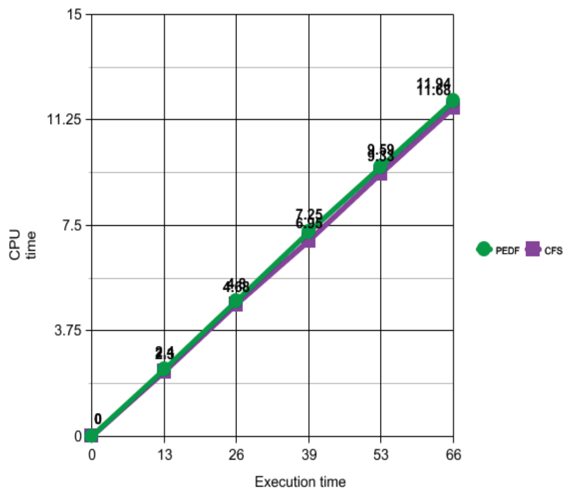

Fig. 5 CPU execution time of a task in a pinned processor framework

## VI. CONCLUSION

The execution overhead caused by the real-time tasks is reduced significantly by executing the real-time and non real-time tasks concurrently in their respective cores namely worker cores and the Linux cores respectively. This prevents the higher priority real-time jobs from preempting the execution of low priority Linux jobs. The proposed method will reduce the execution overhead of the real-time task, since the CPU cores are partitioned in such a way that CPU core 0 is allotted to Linux core for handling Linux tasks, CPU core 1 and core 2 are allotted to the worker cores for handling the real-time tasks and the master core is pinned to CPU core 3 where the scheduling policies for the real-time tasks are implemented. Thus by partitioning the cores between the real-time and non real-time tasks, the real-time and the non real-time tasks are executed in parallel without one interrupting another due to their associated priorities. Furthermore the Linux interrupts are redirected to Linux core that is the core 0 to further reduce the execution overhead of hard real-time tasks. Partitioned Earliest Deadline First scheduling is used for the real-time task scheduling. The proposed method handles both the hardware as well as the software interrupts. Thus the execution overhead of the hard real-time tasks is further reduced by interrupt handling through partitioned scheduling.

## REFERENCES

[1] N. Saranya and R. C. Hansdah, “An Implementation of Partitioned Scheduling Scheme for Hard Real-Time Tasks in Multicore Linux with Fair Share for Linux Tasks,” in Proc. Of the Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014, pp. 1–9.
[2] Brosky. S, Rotolo. S, “Shielded processors: guaranteeing sub-millisecond response in standard Linux,” in Proc. Parallel and Distributed Processing Symposium, p.9, 2003.
[3] S. Brosky, “Shielded cpus: real-time performance in standard linux,” Linux Journal, vol. 121, no. 9, p. 21, 2004.
[4] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, “Litmusˆ rt: A testbed for empirically comparing real-time multiprocessor schedulers,” in Real-Time Systems Symposium, 2006.RTSS’06. 27th IEEE International. IEEE, 2006, pp. 111–126.
[5] B. B. Brandenburg and J. H. Anderson, “On the implementation of global real-time schedulers,” in Real-Time Systems Symposium, 2009,RTSS 2009. 30th IEEE. IEEE, 2009, pp. 214–224.
[6] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, “An edf scheduling class for the linux kernel,” in Proc. of the Real-Time LinuxWorkshop, 2009.
[7] Lakshmanan. K,Kato. S,Rajkumar.R, “Scheduling Parallel Real-Time Tasks on Multi-core Processors,” in proc. Of the Real-Time System Symposium (RTSS), 2010, pp. 259–268.
[8] A. Bastoni, B. B. Brandenburg, and J. H. Anderson,“An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers,” in Real-Time Systems Symposium (RTSS), 2010 IEEE 31st. IEEE, 2010, pp. 14–24.
[9] Y. Zhang, N. Guan, Y. Xiao, and W. Yi,“Implementation and empirical comparison of partitioning-based multi-core scheduling,” in IndustrialEmbedded Systems (SIES), 2011 6th IEEE International Symposium on.IEEE, 2011, pp.248–255.
[10] J. Lelli, G. Lipari, D. Faggioli, and T. Cucinotta, “An efficient and scalable implementation of global edf in linux,” in Proceedings of theinternational workshop on operating systems platforms for embedded real-time applications (OSPERT), 2011.
[11] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, “Is semi-partitioned scheduling practical?” in Real-Time Systems (ECRTS), 2011 23rd Eu-romicro Conference on. IEEE, 2011, pp. 125–135.

[12] M. Asberg, T. Nolte, S. Kato, and R. Rajkumar,"Exsched: an external cpu scheduler framework for real-time systems," in Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on. IEEE, 2012, pp. 240–249.
[13] Kim Hyungwoo, Yoon Hyunmin, Wu Peng, Ryu Minsoo, "Fixed Share Scheduling via Dynamic Weight Adjustment in Proportional Share Scheduling Systems," in Proc. of the Foundation of Computer Science, 2014.
[14] Mikael Asberg, Thomas Nolte and Shinpei Kato, "Towards Partitioned Hierarchical Real-Time Scheduling on Multi-core Processors," in ACM SIGBED, Volume 11 Issue 2, 2014, pp. 13-18.
[15] Cerqueira. F,Vanga. M,Brandenburg. B.B, "Scaling Global Scheduling with Message Passing," in Proc. Of the Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014, pp. 263–274.
[16] Saranya. N,Hansdah. R.C, "Dynamic Partitioning Based Scheduling of Real-Time Tasks in Multicore Processors," in Proc. Of Real-Time Distributed Computing (ISORC), 2015, pp. 190–197.
[17] Tudor David, Rachid Guerraoui, Vasileios Trignoakis,"Everything you always wanted to know about synchronization but were afraid to ask," in Proc. Of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013, pp. 33–48.