Process Scheduling Technique for Multiprocessing

Dr. Swapan Debbarma
Assistant Professor, Department of CSE,
NIT Agartala, Jirania, Tripura

Abstract: Previously, several works have been carried out on hierarchical scheduling frameworks from a theoretical point of view. In the recent past, hierarchical scheduling frameworks have been implemented on uniprocessor system. In this paper I present my initial proposal of the implementation of my hierarchical scheduling framework in a commercial operating system like Vx Works for Symmetric Multiprocessing based (SMP) Real Time Systems. In this paper I perform analysis of the previous implementations and demonstrate feasibility of my implementation in a commercial operating system, without having to modify the kernel source code. I present the SMP structure of Works and how it can be used by extension of previous designs of hierarchical scheduling framework.

Index Terms **– Multicore, real time, Symmetric multiprocessing, VxWorks.**

I. INTRODUCTION

Microprocessor technology is rapidly developing, therefore, multiprocessor and multi-core designs are becoming an attractive solution to fulfil increasing performance demands.

In the real-time systems community, there has been a growing interest in real-time multiprocessor scheduling theories. Multi-core architectures have received significant interest as thermal and power consumption problems limit further increase of speed in single-cores. In the multi-core research community a considerable amount of work has been done on real-time multi-core scheduling algorithms where it is assumed tasks are independent. In general, existing real-time scheduling approaches over „m" processors can fall into two categories: partitioned and global scheduling. Each task under partitioned scheduling is statically assigned to a single processor and is allowed to execute on that processor only. Under global scheduling, tasks are allowed to dynamically migrate across m processors and execute on any of them.

Hierarchical scheduling has shown to be a useful mechanism in supporting modularity of real-time software by providing temporal partitioning among applications. In hierarchical scheduling, a system can be hierarchically divided into a number of subsystems that are scheduled by a global (system-level) scheduler. Each subsystem contains a set of tasks that are scheduled by a local (subsystem-level) scheduler. The Hierarchical Scheduling Framework (HSF) allows for a subsystem to be developed and analysed in isolation, with its own local scheduler, and then at a later stage, using an arbitrary global scheduler, it allows for the integration of multiple subsystems without violating the temporal properties of the individual subsystems analysed in isolation. The integration involves a system level schedulability test, verifying that all timing requirements are met. Hence, hierarchical scheduling frameworks naturally support concurrent development of subsystems. My overall goal is to make hierarchical scheduling a cost-efficient approach applicable for a wide domain of applications, including automotive, automation, aerospace and consumer electronics.

Over the years, there has been a growing attention to HSFs for real-time systems. Since a two-level HSF [1] has been introduced for open environments, many studies have been proposed for its schedulability analysis of HSFs [2]-[3]. Various processor models, such as bounded-delay [4] and periodic [5], have been proposed for multi-level HSFs, and schedulability analysis techniques have been developed for the proposed processor models [6]-[12]. Recent studies have been introduced for supporting logical resource sharing in HSFs [13]-[15]. However, those studies have worked on various aspects of HSFs from a theoretical point of view up until now. This paper presents my work towards a full implementation of a hierarchical scheduling framework.

I have chosen to implement it in a commercial operating system like VxWorks, since there is plenty of industrial embedded software available, which can run in the hierarchical scheduling framework.

II. RELATED WORK

Related works have recently implemented different schedulers in commercial real-time operating systems, where it is not feasible to implement the scheduler directly inside the kernel (as the kernel source code is not available). Also, some work related to efficient implementations of schedulers are outlined in this paper. Many partitioning algorithms and global scheduling algorithms have been proposed in the past. Partitioning algorithms which allow a task to execute on at most two processors have been proposed in the past by Andersson *et al.* [15] and Kato and Yamasaki [16]. These studies focus on implicit-deadline task. In another direction, global scheduling algorithms, US-EDF[m/2m−1] [17] and fp-EDF [18], which give special treatment to certain high utilization tasks have also been proposed. Baruah and Carpenter [19] introduced an approach that can restrict the processor migration of jobs, in order to alleviate the inflexibility of partitioned scheduling and the processor migration overhead of global scheduling. Their approach can be categorized as job-level partitioned scheduling. They shoId that task-level and job-level partitioned scheduling approaches do not dominate each other. Behnam [20] has implemented HSF on VxWorks for uniprocessor systems. The work presented in this paper differs from the last work in [20] in the sense that it implements a hierarchical scheduling

framework in a commercial operating system like VxWorks for multicore systems. Furthermore, the work can be considered as an extension from the above approaches in the sense that it implements a hierarchical scheduling framework intended for open environments [1], and Symmetric Multiprocessing (SMP) systems where real-time applications may be developed independently and unaware of each other and still there should be no problems in the integration of these applications into one environment. A key here is the use of Ill-defined interfaces representing the collective resource requirements by an application, rich enough to allow for integration with an arbitrary set of other applications without having to redo any kind of application internal analysis.

III. SMP ON VXWORKS

VxWorks SMP is a configuration of VxWorks designed for symmetric multiprocessing (SMP). It provides the same distinguishing RTOS characteristics of performance and determinism as the uniprocessor (UP) configuration. The differences between the SMP and UP configurations are limited, and strictly related to support for multiprocessing. Multiprocessing systems include two or more processors in a single system. Symmetric multiprocessing (SMP) is a variant of multiprocessing technology in which one instance of an operating system controls all processors, and in which memory is shared. SMP differs from asymmetric multiprocessing (AMP) in that an AMP system has a separate instance of an operating system executing on each processor (and each instance may or may not be the same type of operating system). My proposed work is restricted to SMP only. The terms CPU and processor are often used interchangeably in computer documentation. However, it is useful to distinguish between the two for hardware that supports SMP. In the context of VxWorks SMP, the terms are used as follows [21]:

CPU: A single processing entity capable of executing program instructions and processing data (also referred to as a core, as in multicore).

Processor: A silicon unit that contains one or more CPUs.

Multiprocessor: A single hardware system with two or more processors.

Uniprocessor: A silicon unit that contains a single CPU. With few exceptions, the SMP and uniprocessor (UP) configurations of VxWorks share the same API--the difference amounts to only a few routines. A few uniprocessor APIs are not suitable for an SMP system, and they are therefore not provided. Similarly, SMP-specific APIs are not relevant to a uniprocessor system--but default to appropriate uniprocessor behaviours (such as task spinlocks defaulting to task locking), or have no effect. VxWorks SMP is designed for symmetric target hardware. That is, each CPU has equivalent access to all memory and all devices as shown in Fig. 1. [21]. VxWorks SMP can therefore run on targets with multiple single-core processors or with multicore processors, as long as they provide uniform memory access (UMA) architecture with hardware managed cache-coherency.

Regardless of the number of CPUs (typically 2, 4 or 8) in an SMP system, the important characteristics are the same:

1. Each CPU accesses the very same physical memory subsystem; there is no memory local to a CPU. This means it is irrelevant which CPU executes code.
2. Each CPU has its own memory management unit that allows concurrent execution of tasks with different virtual memory contexts.
3. Each CPU has access to all devices. Interrupts from these devices can be routed to any one of the CPUs through a programmable interrupt controller. This means that it is irrelevant which CPU executes interrupt service routines (ISRs) when handling interrupts.
4. Tasks and ISRs can be synchronized across CPUs and mutual exclusion enforced by using spinlocks.Bus snooping logic ensures the data caches between CPUs are always coherent. This means that the operating system does not normally need to perform special data cache operations in order to maintain coherent caches. However, this implies that only memory access attributes that allow bus snooping are used in the system. Restrictions in terms of memory access modes allowed in an SMP system, if any, are specific to hardware architecture.

IV. IMPLEMENTATION

VxWorks offers two different modes for application-tasks to execute; either kernel mode or user mode. In kernel mode, application-tasks can access the hardware resources directly. In user mode, on the other hand, tasks cannot directly access hardware resources, which provide greater protection (e.g., in user mode, tasks cannot crash the kernel). Kernel mode is provided in all versions of VxWorks while user mode was provided as a part of the Real Time Process (RTP) model, and it has been introduced with VxWorks version 6.0 and beyond.

In this paper, I are considering kernel mode tasks since such a design would be compatible with all versions of VxWorks I are also considering fixed priority preemptive scheduling policy for the kernel scheduler (not the round robin scheduler). A tasks priority should be set when the task is created, and the task's priority can be changed during the execution. Then, during runtime, the highest priority ready task will always execute. If a task with priority higher than that of the running task becomes ready to execute, then the scheduler stops the execution of the running task and instead executes the one with higher priority. When the running task finishes its execution, the task with the highest priority among the ready tasks will execute.

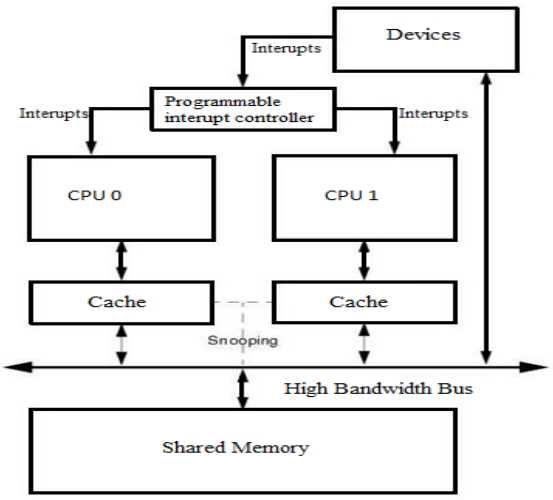

Fig. 1. SMP on VxWorks

Periodic Tasks

In this paper I consider a periodic task model, $\tau_i(T_i,C_i,D_i)$ similar to [20]. Where T_i is the task period, C_i is a worst-case execution time requirement, and D_i is a relative deadline ($0 < C_i \leq D_i \leq T_i$). The set of all tasks is denoted by Γ($\Gamma = \{\tau_i \mid$ for all i= 1, .., n}where n is the number of tasks). A task that becomes ready for execution periodically once every n^{th} time unit which means a new instant of the task is executed every constant period of time is a periodic task. VxWorks including most commercial operating systems do not directly support the periodic task model [22]. A periodic task can be implemented by allowing a task to sleep until the beginning of its next period when a task finishes its execution. Such periodic behavior can be implemented in the task by the usage of timers. An important point here is that a task typically does not finish its execution at the same time always, as execution times and response times vary from one period to another. Hence, whenever a task finishes its execution use of timers may not be easy and accurate as the task needs to evaluate the time for next period relative to the current time. The reason for this is that preemption may happen between the time measurement and calling the sleep function. Support for periodic activation of servers is required in order to implement the hierarchical scheduling framework. My hierarchical scheduling framework is based around the periodic resource model [5] and [20]. Similar to [20], 23]-[24] , I have implemented the periodic resource model by the usage of a server based approach that replenish their budget every constant period, i.e., the servers behave like periodic tasks.

Supporting Arbitrary Schedulers

Similar to [20] in this paper I propose to use an Interrupt Service Routine (ISR) to manipulate the tasks in the ready queue. Apart from changing their priorities, the ISR is responsible for adding tasks in the ready queue according to the hierarchical scheduling policy in use. I would prefer to use User Scheduling Routine (USR) instead of ISR in rest of the paper. Any desired scheduling policy can be implemented by using the USR including the much common types such as Rate Monotonic (RM) and Earliest Deadline First (EDF).

Hierarchical Scheduling Implementation

There exist (to the best of my knowledge) four known hierarchical multi-core scheduling schemes for real-time systems, where one of these has been implemented in Linux. Shin *et al.* [25] has proposed analysis for cluster-based scheduling. In this setup, tasks are scheduled with G-EDF (global EDF) at the local level, i.e., the ready queue is ordered by shortest deadline and tasks are transferred to the different cores until they are all occupied. At the global level, each subsystem, i.e., group of tasks and servers are assigned a subset of the cores. The global scheduler schedules the subsystems servers based on G-EDF and the assigned cores. The subsystem has the same amount of servers as the amount of assigned cores, and all servers have the same period, but the budget is distributed among them. This distribution is not fair, e.g., 3 servers with budget 2.5 will result in budgets 1, 1 and 0.5. In Checconi *et al.* [26] (has an implementation in Linux), each subsystem will have access to all cores, and the number of servers (in each subsystem) are the same as the number of cores. Locally, tasks are scheduled with global multi-core fixed priority, globally, each core has its own H-CBS (Hard Constant Bandwidth Server) scheduler, scheduling one server from each subsystem. Note that the global scheduling is done in parallel (several H-CBS schedulers). Nemati *et al.* [27] has a scheme where the servers are scheduled with a global multi-core scheduling scheme (fixed or dynamic priority), and locally, each subsystem is scheduled with partitioned multi-core (fixed or dynamic priority) scheduling, i.e., each subsystem has maximum one server (that may run on any core), so tasks always execute on the same core. A fourth scheme [28] could be to schedule servers in sequence, thereby scheduling tasks, within each subsystem, with global multi-core on all cores. The difference from [26] is that there is only one global scheduler. In Shin *et al.* [25], the servers of a subsystem typically occupies a subset of all cores at a time (depending on the priority of other subsystems and its assigned number of cores). Checconi *et al.* [26] differs in that there is one scheduler per core, scheduling independently of one another. A subsystem has one server per core with the same parameters. In Nemati *et al.* [27],

there is maximum one subsystem server running at once, but it may migrate, as illustrated. My proposed scheme, the sequential approach, is an extension of [20] and [28-33], and it will execute the subsystems in sequential order, occupying each core with one server on the VxWorks system. The Global scheduling layer supports both FPS and EDF scheduling together with the periodic server model. Global scheduler is responsible for handling all events in the system which can be local task sub-system events, server period events or server budget events. Local scheduler is not aware of the fact that it schedules different task systems; its interface to Global Scheduler is a server which has its own task *tcb* and task TEQ's (Total Event Queue). TEQ is a sorted queue where the USR stores the next activation time of all tasks (absolute times) according to the closest time event. Server periods are stored in the server event queue including a reference to its associated server. The server event queue is examined in the case of a server period interrupt execution. The server period events can easily be handled due to the reference to the server TCB. The server TCB is referenced into the server ready queue when the server has a period start. Those tasks that are in the VxWorks ready queue have a reference from the queue to its TCB. Each node in the *period_event_queue* and *deadline_event_queue* (Fig. 2) has each a reference to its associated tasks TCB. In the case of task period interrupt execution, the TCB can easily be referenced and put in the task ready queue. The global scheduler chooses the closest event of the running servers task deadline TEQ, the task period TEQ, the server budget TEQ (only one node) and the server period TEQ (I assume that server period is equal to server deadline). In Fig. 2, the next event would be a task, (*task1* in this case) if server 1 was the active server at that point. Fig. 2 also illustrates that each server has a reference to a part of the VxWorks task ready queue; this is hidden from the local scheduler which schedules as if this is the whole task *tcb* of the system. Each server in the ready queue is sequentially assigned to a core. There will be separate VxWorks ready queues for each core in the system. The allotment of the ready queues is not fixed to any particular core.

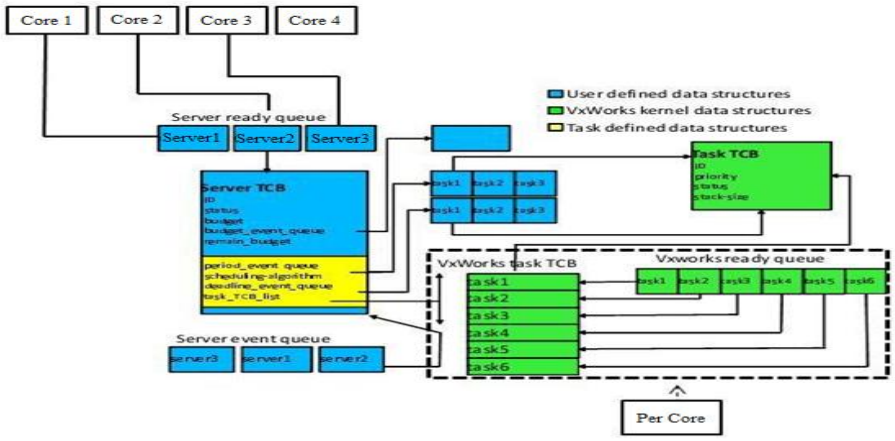

Fig. 2. HSF implementation on VxWorks SMP

Example

Fig. 3 illustrates the implementation of hierarchical scheduling framework on a single processor whereas Fig. 4 illustrates my modified implementation which includes an example with three servers *S*1, *S*2, *S*3 with global and local RM schedulers, the priority of S1 is the highest and the priority of *S*3 is the lowest. Suppose a new period of *S*2 and *S*3 start at time *t*0 and *t*1with a budget equal to *Q*3. Then, the USR will change the state of *S*2 and *S*3 to Ready, and

1. Add the time at which the budget will expire, which equals to *t*0 + *Q*3, *t*1 + *Q*3 for *S*2 and *S*3 respectively into the server event queue and also add the next period event in the server event queue.
2. Check all previous events that have occurred while the server was not active by checking if there are task releases or deadline checks in the time interval of [*t*∗, *t*0], [*t*1∗, *t*1], where t0∗ and *t*1* are the latest time at which the budget of S2 and S3 have been expired.
3. Start the local scheduler.
   At time t1 the server S2 becomes ready and it has higher priority than S3. Unlike in [20], S2 will not preempt S3 and, the USR will not remove tasks that belong to S3 from the ready queue. Instead, S3 will continue on the core to which it is attached and S2 will be assigned a different core. S2 will preempt S3 only if no free core is available at that time.

V. CONCLUSION

This paper has presented my initial analysis and implementation of a hierarchical scheduling framework on a commercial operating system like VxWorks for SMP hardware. Although previous works on similar lines [20], have been proposed, I have tried to improvise them in my work by extending their implementation on SMP architecture. The decision to choose VxWorks as the target Operating System is to understand and explore the intricacies of such an industry dominant operating system. Another motivation for my work is to develop case studies both for academia and industry. I are able to identify that a hierarchical scheduling framework can effectively be implemented on multicore architectures with some overheads and can achieve the clean separation of subsystems in terms of timing characteristics. However, my proposed implementation does not consider the utilization of the cores with resource sharing. There is tremendous amount of future scope for my work that includes support for sporadic as Ill as aperiodic tasks. Sporadic tasks can be supported in response to specific events such as external interrupts.

Support for aperiodic tasks could be done by bounding their interference to periodic tasks by the use of some server based mechanisms. Future implementation may also include support for AMP based systems.

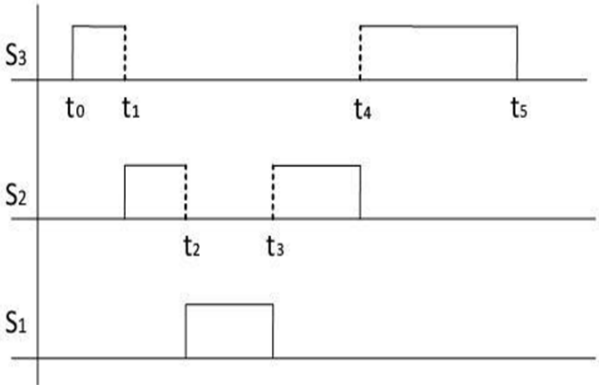

Fig. 3. Simple servers execution example on single core

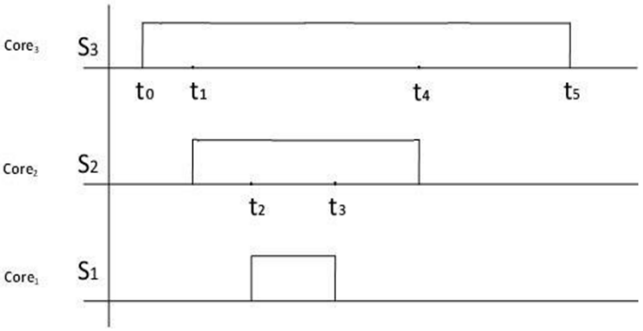

Fig. 4. Simple servers execution example on multicore

REFERENCES

[1] Z. Deng and J. W. S. Liu, "Scheduling real-time applications in an open environment," in Proc. of IEEE Real-Time Systems Symposium, pp. 308–319, December 1997.

[2] T. W. Kuo and C. Li, "A fixed-priority-driven open environ-ment for real-time applications," in Proc. of IEEE Real-Time Systems Symposium, pp. 256–267, December 1999.

[3] G. Lipari and S. Baruah, "Efficient scheduling of real-time multi-task applications in dynamic systems," in Proc. Of IEEE Real-Time Technology and Applications Symposium, pp. 166–175, May 2000.

[4] A. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in Proc. of IEEE Real-Time Technology and Applications Symposium, pp. 75–84, May 2001.

[5] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in Proc. of IEEE Real-Time Systems Symposium, pp. 2–13, December 2003.

[6] L. Almeida and P. Pedreiras, "Scheduling within temporal partitions: response-time analysis and server design," in Proc. of the Fourth ACM International Conference on Embedded Software, September 2004.

[7] R. I. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in Proc. of the 26th IEEE International Real-Time Systems Symposium (RTSS'05), December 2005.

[8] X. Feng and A. Mok, "A model of hierarchical real-time virtual resources," in Proc. of IEEE Real-Time Systems Symposium, pp. 26–35, December 2002.

[9] G. Lipari and E. Bini, "Resource partitioning among real-time applications," in Proc. of Euromicro Conference on Real-Time Systems, July 2003.

[10] S. Saewong, R. Rajkumar, J. Lehoczky, and M. Klein, "Anal-ysis of hierarchical fixed-priority scheduling," in Proc. of Euromicro Conference on Real-Time Systems, June 2002.

[11] I. Shin and I. Lee, "Compositional real-time scheduling framework," in Proc. of IEEE Real-Time Systems Sympo-sium, December 2004.

[12] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems," in Proc. of the 7th ACM and IEEE International Conference on Embedded Software (EMSOFT'07), 2007.

[13] R. I. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in Proc. of the 27th IEEE International Real-Time Systems Symposium (RTSS'06), December 2005.

[14] N. Fisher, M. Bertogna, and S. Baruah, "The design of an EDF-scheduled resource-sharing open environment," in Proc. of the 28th IEEE International Real-Time Systems Symposium (RTSS'07), pp. 83–92, December 2007.

[15] B. Andersson and E. Tovar, Multiprocessor scheduling with few preemptions, InRTCSA, 2006.

[16] S. Kato and N. Yamasaki, Real-time scheduling with task splitting on multiprocessors, InRTCSA, 2007.

[17] A. Srinivasan and S. Baruah, "Deadline-based scheduling of periodic task systems on multiprocessors," Information Processing Letters, vol. 84, no. 2, pp. 93–98, 2002.

[18] S. Baruah, "Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors," IEEE Trans. on Computers, vol. 53, no. 6, pp. 781–784, 2004.

[19] S. K. Baruah and J. Carpenter, Multiprocessor fixed-priority scheduling with restricted interprocessor migrations, InECRTS, 2003.

[20] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. Bril, Towards Hierarchical Scheduling on top of VxWorks, MRTC Mälardalen University, Technische Universiteit Eindhoven, Västerås, Eindhoven}, SIden, Holland, 2008.

[21] VxWorks Kernel Programmer's Guide pp. 6-8, 2009.

[22] J. Liu, Real-time systems , Prentice Hall, 2000.
[23] J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," in Proc. of 8th IEEE International Real-Time Systems Symposium (RTSS'87), pp. 261–270.
[24] B. Sprunt, L. Sha, and J. P. Lehoczky, "Aperiodic task scheduling for hard real-time systems," Real-Time Systems, vol. 1, no. 1, pp. 27–60, June 1989.
[25] I. Shin, A. Easwaran, and I. Lee, "Hierarchical Scheduling Framework for Virtual Clustering of Multiprocessors," in Proc. of the 20th Euromicro Conference on Real-Time Systems, July 2008.
[26] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari, "Hierarchical Multiprocessor CPU Reservations for the Linux Kernel," in Proc. Of the 5th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, June 2009.
[27] F. Nemati, M. Behnam, and T. Nolte, "Multiprocessor Synchronization and Hierarchical Scheduling," in Proc. of the 1st International Workshop on Real-time Systems on Multicore Platforms: Theory and Practice, in conjunction with ICPP'09, September 2009.
[28] M. Åsberg, T. Nolte, and S. Kato, Towards Hierarchical Scheduling in Linux/Multi-core Plateform, MRTC Mälardalen University, Technische Universiteit Eindhoven, SIden, The University of Tokyo,2010.
[29] Pradhan, M. C., Satpathy, S., & Bhoi, B. K. (2015, May). An improved FPGA based model for automatic traffic sign detection. In 2015 International Conference on Smart Technologies and Management for Computing, Communication, Controls, Energy and Materials (ICSTM) (pp. 291-298). IEEE.
[30] Pradhan, M. C., Satpathy, S., & Bhoi, B. K. (2016). An Intelligent Fuzzy Based Technique of Making Food Using Rice Cooker. Asian Journal of Electrical Sciences, 5(1), 1-7.
[31] Satpathy, S., Das, S., & Debbarma, S. (2019). Development a Novel Approach of Fuzzy Based FPGA System for Prediction of Jaundice in Rural Area. Journal of Engineering Technology (ISSN. 0747-9964), 8(1), 32-39.
[32] Sengupta, A. S., Satpathy, S., Mohanty, S. P., Baral, D., & Bhattacharyya, B. K. (2018). Supercapacitors Outperform Conventional Batteries [Energy and Security]. IEEE Consumer Electronics Magazine, 7(5), 50-53.
[33] Satpathy, S., & Sahu, A. P. (2015, December). A Graphical User Interface, Fuzzy Based Intelligent Rice Cooker. In 2015 International Conference on Computational Intelligence and Communication Networks (CICN) (pp. 1216-1220). IEEE