AN EFFICIENT HYBRID SCHEDULER USING DYNAMIC SLACK FOR REAL-TIME CRITICAL TASK SCHEDULING IN MULTICORE AUTOMOTIVE ECUs

Geetishree Mishra[1], Maanasa RamPrasad[2], Ashwin Itagi[3] and K S Gurumurthy[4]

[1]BMS College of Engineering, Bangalore, India,
[2]ABB India Limited, Bangalore, India
[3]CISCO Systems, Bangalore, India
[4]Reva Intitute of Technology, Bangalore, India

ABSTRACT:

Task intensive electronic control units (ECUs) in automotive domain, equipped with multicore processors , real time operating systems (RTOSs) and various application software, should perform efficiently and time deterministically. The parallel computational capability offered by this multicore hardware can only be exploited and utilized if the ECU application software is parallelized. Having provided with such parallelized software, the real time operating system scheduler component should schedule the time critical tasks so that, all the computational cores are utilized to a greater extent and the safety critical deadlines are met. As original equipment manufacturers (OEMs) are always motivated towards adding more sophisticated features to the existing ECUs, a large number of task sets can be effectively scheduled for execution within the bounded time limits. In this paper, a hybrid scheduling algorithm has been proposed, that meticulously calculates the running slack of every task and estimates the probability of meeting deadline either being in the same partitioned queue or by migrating to another. This algorithm was run and tested using a scheduling simulator with different real time task models of periodic tasks . This algorithm was also compared with the existing static priority scheduler, which is suggested by Automotive Open Systems Architecture (AUTOSAR). The performance parameters considered here are, the % of core utilization, average response time and task deadline missing rate. It has been verified that, this proposed algorithm has considerable improvements over the existing partitioned static priority scheduler based on each performance parameter mentioned above.

KEY WORDS

ECU, Multicore, RTOS, Scheduling, OEM, AUTOSAR.

1.INTRODUCTION

Automotive electronic subsystems are the real time embedded systems expected to work efficiently at various operating environments adhering to the safety critical requirements and strict government regulations. Within short regular intervals, new advanced and complex features are getting added to the existing systems. Original Equipment Manufacturers (OEMs) are competing with each other to introduce more sophisticated applications to their automobiles [1]. Multicore processors are the driving architectures for these new evolutions in automotive domain [2]. In this context, Automotive Open System Architecture (AUTOSAR) supported OEMs to add more functionality into the existing ECUs and shift out from one function per ECU paradigm. AUTOSAR suggested more centralized software architecture designs with appropriate protection mechanisms. Due to this, high computational capabilities were expected from the ECU processor. For several years there has been a steady rise in the amount of computational power required from an ECU with architectural changes in the hardware and by increases in clock rate. But increase in clock rate also increases power consumption and also creates complex electromagnetic compatibility (EMC) issues. High performance and low power consumption being the two classic, conflicting requirements of any embedded system design, further frequency scaling the processor for achieving high performance was not a viable option any more. In this consequence, multicore processors were introduced in ECUs as a cost effective solution to achieve greater performance at moderate clock speed and low power consumption [2]. Because of multicore implementation, the ECUs are computationally more capable of accommodating much functionality which is an opportunity for OEMs to reduce the number of ECUs, the complex network connections and communication buses. Multicore ECUs bring major improvements for some applications that require high performance such as high-end engine controllers, electric and hybrid powertrains and advanced driver assistance systems which sometimes involve realtime image processing [1]. The inherent parallelism offered by multicore platforms helps in segregating the safety critical functions and allocating to dedicated cores. The challenges lie in the software architecture design which is also required to be parallelized to exploit the maximum capability of parallelized hardware. Decomposition of the application tasks and interrupt handlers into independent threads is required to achieve parallel execution by multiple cores [3,4]. To optimize the performance of an application running on a multicore platform, real-time CPU scheduling algorithms play the pivotal role. There are many existing methods and algorithms of real time task scheduling which are required to be reanalyzed and modified to suit to the parallel hardware and software architecture [5]. AUTOSAR has suggested strictly partitioned and static priority scheduling for safety critical tasks in automotive domain but there are issues like partitioning the tasks and mapping to a fixed core [2]. During heavy load condition, the lower priority tasks always starve for a computing resource and are sometimes forced to miss their deadlines. With the motivation of deriving a suitable task scheduling method and algorithm for multicore ECUs, a hybrid scheduling model and an algorithm have been proposed in this work. A simulation process with different task models has also been explained. The simulation results with different performance parameters are demonstrated. The paper is organized as: section II, introduces the existing task scheduling scenario in multicore automotive ECUs. section III explains about the proposed scheduler model for a tricore ECU. In section IV the task model and schedulability conditions are derived., In section V the proposed algorithm is presented, Section VI provides an illustration on the results and discussions. Section VII presents the performance analysis and Section VIII gives the conclusion of the work.

2.EXISTING SCHEDULING SCENARIO IN AUTOMOTIVE ECUs

AUTOSAR version 4.0 suggests preemptive static priority and partitioned scheduling for safety critical tasks on multicore implemented ECUs [2]. In a current implementation, there are three computational cores of a multicore processor used for the diesel gasoline engine control ECU. The peripheral core, the performance core and the main core. The peripheral core is mostly a DSP core, lock step mode is implemented in the main core to protect data consistency and the performance core which is mostly kept as a support system otherwise used for periodic tasks in high load conditions.

2.1 Task Partitioning

There are different task distribution strategies used to partition the tasks to achieve deterministic behaviour of the system [6,7,8]. In one of the strategies, basic system software (BSW) and complex drivers are allocated to the peripheral core. Crank teeth angle synchronous tasks of application software (ASW) and monitoring relevant tasks are allocated to the main core [9]. Time triggered tasks of ASW are allocated to the performance core. The preemptive, non-preemptive and cooperative tasks are segregated and allocated to different cores. Similarly in another implementation strategy, tasks and interrupt clusters are defined based on the hardware dependencies, data dependencies and imposed constraints [2,10]. All the tasks in any cluster are allocated to the same core to be executed together to avoid the communication costs. An example tasks/interrupts clusters are shown in the fig.1 below. Each task of a task cluster is mapped to an OS Application. The OS Application is then mapped to a core by OS configuration [11]. Within a task, there will be a chain of nunnables to be executed sequentially. Various runnable sequencing algorithms are there in the literature to generate a proper flow of execution, but the performance evaluation of such distribution is not satisfactory because it is difficult to partition between tasks, ISRs and BSW with high load conditions [10]. As a result, there is poor load balancing between the cores in heavy load conditions. Furthermore there cannot be a fixed task distribution strategy for all the systems as each system has its own software design. There is a need of trade-off between flexibility and effort in task distribution strategy. In this proposed algorithm, both global and partitioned queues are used and task migrations are allowed to explore the availability of computing cores.

2.2 Static priority Scheduling

AUTOSAR suggests static priority cyclic scheduling for the tasks in every partitioned queue [2]. The number of runnables grouped under a task is always large. The runnables under OS services or application tasks with same periodicity or ISRs of same category are grouped together. In the static priority scheduling, priorities are assigned to each task prior to run time. Each task is defined with four parameter values: initial offset, worst case execution time (WCET), its period and priority. At any releasing instant at the partitioned queue, the higher priority task get scheduled for execution. So at higher load conditions, low priority tasks do not get a CPU slot and because of strict partitioning, they are unable to migrate to other core even though the consequence is missing the deadlines [1,11]. CPU utilization is also not balanced with task partitioning. So there is a scope for dynamic task scheduling as well [12]. This research work is intended to address this issue and an effort has been made to develop an efficient hybrid task scheduling algorithm for a multicore ECU which is tested with various open source scheduling tools.

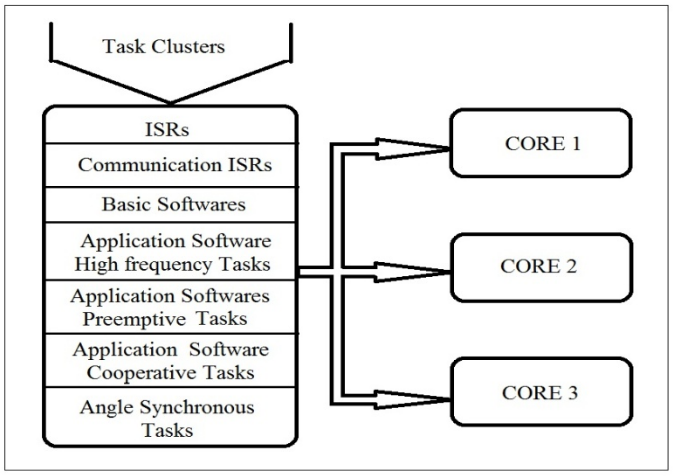
Task Clusters
ISRs
Communication ISRs
Basic Softwares
Application Software High frequency Tasks
Application Softwares Preemptive Tasks
Application Software Cooperative Tasks
Angle Synchronous Tasks
CORE 1
CORE 2
CORE 3

Fig.1 Task Clusters

3.PROPOSED SCHEDULER MODEL

The scheduler model is characterised by a global queue, three partitioned queues serving for three identical cores and an intermediate module called task distributer. The model refers to a task parameters table and based on each activation instant, the tasks get added to the global queue. The tasks distributer is an interface between the global and the partitioned queues that takes the decision on which task from the global queue to move to which local queue based on the calculated slack and number of tasks at the global queue. The hybrid scheduling algorithm is implemented as a scheduler function and runs at every time quantum which is fixed for a task model and runs on one of the cores.

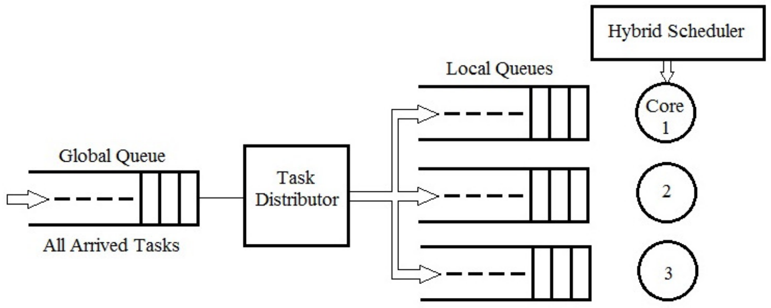

Fig.2 Scheduler model

4. TASK MODEL DESCRIPTION

In this paper, we have considered a set of 10 periodic tasks of application software for engine control ECU that consists of three identical cores. In practice, the application software is decomposed into tasks and many processes run within a task. A task is called from the OS kernel whenever appropriate. In an engine control unit two types of tasks are executed: the asynchronous or time-triggered tasks, which are activated periodically and the synchronous or engine-triggered tasks, which are activated at a specific angular position of the engine crank shaft

[9]. As a consequence the frequency of engine-triggered task arrival varies with the speed of the engine. Additionally, the execution time of some of the time triggered tasks may also depend on the speed of the engine. In this paper the proposed hybrid scheduler is tested with three different task models considering the real time behavior of the engine.

4.1 Task Characteristics

The ith task is characterized by a three tuple Ti = (Ci, Ri, Pi). Quantities Ci, Ri, and Pi correspond to the worst case execution time (WCET), the releasing instant and the period [13,14,15]. Subsequent instances of the tasks are released periodically. As shown in fig.3, the next activation instant of task Ti is Ri+Pi. For all the periodic tasks, deadline is equal to the period. Slack of a task is the maximum amount of time it can wait in the queue before execution and still meets its deadline. Slack is calculated for each task and is denoted by Si. Si= Pi – RET; where RET= remaining execution time of a task. In this paper, slack is the utilized parameter to find a feasible schedule.

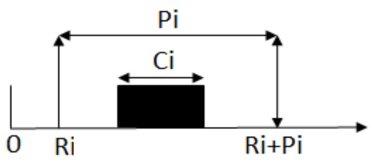

Fig. 3 Task Model

4.1.1Assumptions

In this paper, we have a set of assumptions while creating a task model, considering the automotive applications.

- Each task is periodically executed strictly. The initial releasing instant of a task can be chosen freely within its period with the objective of balancing the central processing unit (CPU) load over a scheduling cycle.
- Periods are chosen as 5ms, 10ms, 20 ms, 50 ms and 100ms for realistic automotive applications.
- The WCET of tasks are randomly chosen to have a CPU utilization of 0.1 to 0.2 assuming that, maximum 5 to 10 tasks will be waiting in the queue for execution at the same instant.
- Slack of a task should be integer multiple of its worst case execution time.
- All cores are identical with regard to their processing speed and hardware features. There are no dependencies between tasks allocated on different cores.
- Tasks are free to migrate across the cores.

4.2 Schedulability Conditions

In this paper, tasks are considered as software components within which a large number of processes can run in a definite sequential order once the task is scheduled. Intra-task parallelism is not considered here. Slack Si of a task is the difference between its period and remaining execution time, is the considered parameter to find out a feasible schedule at each core where all tasks can meet their deadlines.

The schedulability conditions are:

- the total CPU utilization μ by all the n tasks should be less than equal to the number of CPU cores denoted by m [17,18].

  $\mu \leq m$; where m= number of CPU cores.

  $\mu = \sum_{i=1}^{n} Ci/Pi$, where n= number of tasks

- Slack of any task Ti should be an integer multiple of its WCETi.
- Slack Si = Pi – RET , where RET= remaining execution time.
- Remaining slack of a task in a local queue should be greater than the sum of remaining execution time of tasks arranged before it.

5. PROPOSED HYBRID SCHEDULING ALGORITHM

In automotive domain, it is currently the static priority partitioned scheduling used for multicore ECUs. Tasks are strictly partitioned and listed in ready queues from where they are scheduled for execution based on their static priority. At any scheduling instant, the highest priority task in a ready queue runs in the corresponding CPU core. OEMs are following the AUTOSAR suggestions on this aspect looking into the safety criticality of the tasks [9]. The main drawback in this approach is, CPU cores are not utilized properly due to strict partitioning of the tasks. Also at heavy load conditions, low priority tasks are much delayed to get CPU slots for their execution. In this paper, a hybrid scheduling algorithm has been proposed and tested with three different task models using a JAVA based simulation tool for real time multiprocessor scheduling. This algorithm has the features of both global and partitioned scheduling and tasks are allowed to migrate from one core to other with a probability of meeting their deadlines [6,16,17,18]. Slack is the utilized parameter in this algorithm. The task with minimum slack has highest priority.

<u>Hybrid Scheduling Algorithm Pseudocode</u>

```
On activate(Evtcontext)
    {Get(task);
     Compute(slack);                    // Slack= Period-WCET
     Globalqueue.Add(task);}
     Sort(Globalqueue);                 //  Sort in ascending order of slack
OnTick(){
Q=quantum; ntick=0;
ntick=(ntick+1)%Q;
If (ntick==0)
{i=true; j=true;}
Schedule()
{            if (i) {
        slackCompute(Globalqueue);
        laxityCompute(Localqueue);      // Laxity= Slack-Waiting time
         i = false;}
             if (j) {
        Distribute(task);
```

```
             Localqueue.Add(task);
             Globalqueue.remove(task);
             Sort(Localqueue);
               j = false;}}
15  Allocate core(task at Lqueuehead);
16  ET=SetET(T.running);                    // ET=Execution time
17  For(all localqueues){
    (for i=0;i<Lqsize();i++)
    {getET(Ti);
         getWCET(Ti);
      If(ET==WCET)
             {Lqueue.remove(Ti);
               Core.remove(Ti);
               SetET(Ti=0);}}
                           }
18  For(all localqueues){
    If(Trunning != Tqhead){
                     If(laxity(Tqhead)< RET(T.running))
            {Preempt();
              Sort(Lqueue);}}
                           }
19  For(all queues){
    If(Localqueue1.size () >1){
    T=Task.Localqueue1(end);
    X=Compute.laxity (T);}
    If(Localqueue2.size()==0 | Localqueue3.size()==0)
    {Migrate(T);}
    If((Localqueue2.size() > 0) & (Localqueue3.size() > 0)){
    T1 = Task.Localqueue2(head);
    T2 = Task.Localqueue3(head);
    Y=Compute.Laxity(T1);
    Z= Compute.Laxity(T2);}                             k-1
    If((X<Cum_RET.Localqueue1) & (X<Y)) // Cum_RET=  ∑ RET(Ti); ( k=qlength)
    { Migrate(T);                                       i=1
      Sort(Localqueue2);}
    Else If((X<Cum_RET.Localqueue1) & (X<Z))
    {Migrate(T);
    Sort(Localqueue3);}}
```

Fig.4 Hybrid Algorithm

5.1 Task Distribution

The inputs to the task distributor module are the tasks arranged in ascending order of their slack at a releasing instant. As slack is the maximum delay a task can withstand before meeting deadline, it is a constant value at task arrival. So based on the task model under test, the distribution threshold can be set. In this paper, three different automotive system representative task models are tested, for which distribution logics are suitably chosen to balance the load as well as to reduce the number of migration [13,19]. If n is no of tasks arranged in ascending order of their slack at a scheduling instant and if n is an even number, first n/2 tasks to core1, (n-(n/2))/2 to core2 and remaining tasks distributed to core3 while if n is an odd number, (n+1)/2 to core1,(n-(n+1)/2)/2 to core 2 and remaining tasks distributed to core 3. As the number of tasks released and their slacks are independent of each other, in this paper, only number of tasks is considered for task distributions.

5.2 Task Migration

Even though tasks are passed on to the partitioned queues, there are no strict characteristics of tasks and CPU cores to be mapped on one to one. So tasks migration is possible from one queue to other to meet deadlines [20,21]. Migration of tasks is allowed when tasks are going to miss their deadlines waiting in the corresponding queue and there is a probability of meeting deadlines at other queue. Probability of missing deadline in a queue is checked starting from the task at the tail of the queue. If the remaining slack called laxity of the task at the end of the queue is less than the sum of the remaining execution time (RET) of all the tasks waiting in the queue, the migration logic searches for a probability at other queues. As the task distribution logic is so chosen that, more number of tasks with less slacks get accumulated at local queue1, the tasks in this queue look for a migration to other two queues. If the remaining slack of the task at the end of queue1 is less than the same for task at queue head of either of the other queues, then a migration happens from queue1 to queue2 or queue3. In the consequence when queue2 or queue3 is empty, a migration occurs without any slack comparison from more populated queue to the empty queue. In brief these can be presented as:

- When, $Sk < \sum_{i=1}^{k-1} \mathrm{RET(Ti)}$, where Sk= remaining slack of ‘k’th task at the tail of queue1,

  If Sk < Sm | Sk < Sl , where Sm and Sl are remaining slacks at the head of queue2 and queue3 respectively, then task Tk migrates to queue2 or queue3.

- Remaining slack (Ti) = Si - Wt
- Wt (waiting time) = Pt-Ri-ET ; where, Pt=present time (System_tic) Ri = release time of ith task and ET= execution time

6. RESULTS AND DISCUSSIONS

In this paper, the proposed hybrid scheduling algorithm is run and tested on a tricore processor for three different task models, where each model has ten numbers of tasks (m=3 and n=10) denoted as T1 to T10 defined with the parameter attributes defined and explained in section 4.1. Then the currently used partitioned static priority scheduling algorithm was run for the same task models to compare the performances based on core utilization, average response time and missed deadline. The task models used in this paper are the representative task models of automotive applications. To run and validate the algorithms, the JAVA based STORM tool is used which is a scheduling simulator that gives a Gantt chart for each core as the result. The simulator kernel uses an XML file as input, where all the task parameters, the CPU cores, the scheduler class name, the scheduling quantum and also the simulation duration are defined. For clear visibility of results, in this work, simulation duration is taken as 50 ms of CPU time and quantum as 1ms.

6.1 Partition Static Priority Scheduler

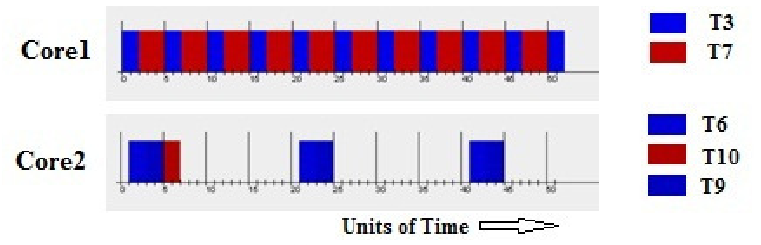

Fig.5 Result Gantt Chart for partition static priority algorithm

The result Gantt charts for a periodic task model for the schedule using partition static priority algorithm are shown in Fig.5. The task model used for simulation has tasks with 5ms, 10ms and 20ms periods. Due to strict partitioning of the tasks, those are with 5ms and 10ms periods allocated to core1 and tasks with 20 ms period are allocated to core2. So from the used task model, seven numbers of tasks are intended for execution in core1 and three numbers of tasks in core2. It can be clearly observed from core1 Gantt chart that, only higher priority tasks T3 and T7 are getting scheduled at the beginning and after a threshold number of times, medium priority tasks T1 and T5 are scheduled with high response time. The low priority tasks T2, T4 and T8 are not getting a schedule slot for execution and are missing deadlines. As only three tasks are allocated to core2, all are meeting their deadlines and core utilization is also very less. Since no partition for core3, it is completely idle and utilization of the core is zero.

6.2 Hybrid Scheduler

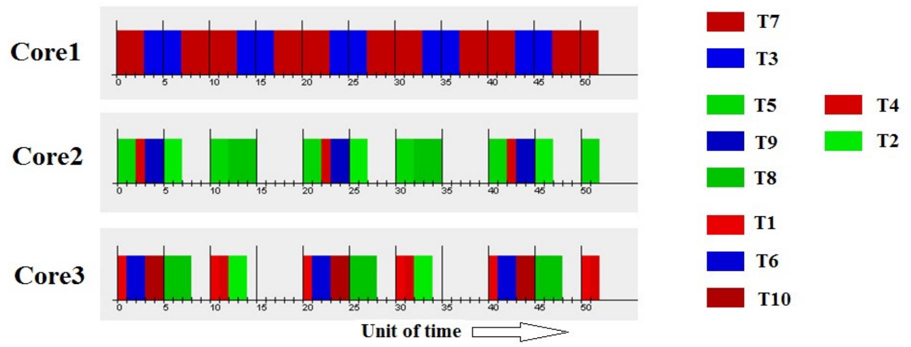

Fig.6 Result Gantt Chart for hybrid scheduling algorithm

The result Gantt charts for the same periodic task model for the schedule using the proposed hybrid scheduling algorithm are shown in Fig.6. As the result widows show, core 1 is fully occupied with task T3 and T7 which have least slack and so highest priorities. At every task releasing instant, after sorting of the tasks in ascending order of their slack at the global queue, the task distribution logic passes atleast 50% of the tasks to local queue1 and as a result, it gets more populated. With the help of its migration logic, the scheduler gets the tasks at the tail of the queue, moved to other queues with the probability of meeting deadlines. The task model used here has total utilization equal to 2.2 which is less than m= 3, the number of CPU cores. As per the schedulability conditions of the algorithm, it can give a feasible schedule if total utilization of the tasks, $\mu \leq$ m. Since the considered task model is relaxed in this respect, the hybrid scheduler gives a feasible schedule for all the tasks and also some idle time slots appear on core2 and core3 Gantt chart. Where partition static priority scheduler does not give a feasible schedule for the task model even with less utilization factor, the proposed hybrid scheduler is able to schedule efficiently with no deadline missing.

7. PERFORMANCE ANALYSIS

In this paper, the parameters considered for comparing the performance of proposed hybrid scheduler with the existing partition static priority scheduler are: the percentage of CPU core utilization, average response time for the scheduled tasks, number of missing deadlines, migration and preemption overheads. The performance of the proposed algorithm based on each parameter for each task model is discussed here in the subsections.

7.1 CPU utilization

CPU Utilization						
	Partition Static Priority Scheduler			Hybrid Scheduler		
Task Model	Core1	Core2	Core3	Core1	Core2	Core3
M1	82%	80%	0	44%	84%	48%
M2	86%	90%	0	82%	56%	54%
M3	100%	28%	0	100%	62%	64%

Table.1 CPU Utilization

The total CPU utilization required by a task model is:

$$\mu = \sum_{i=1}^{n} Ci/Pi, \text{ where n=10}$$

For task model 1, utilization μ is 1.6, for task model 2, μ is 1.75 and for task model 3, it is 2.2. As each task model has run with both the partition and hybrid scheduling algorithms, each has different utilization of CPU cores based on their scheduling strategies. Table 1 gives the utilization percentage of each core for each task model tested on both the algorithms. Utilization of each core is calculated for the simulation duration which is taken as 50ms. As the WCET of each task is taken as an integer value, fractional utilization is not considered. Utilization of each core is the sum of execution time divided by the simulation time. As it is clearly shown in the table 1, for partition static priority scheduling, there is load imbalance across the three cores and no utilization of core3. It happens because of partitioning criteria based on periodicity and interdependency of tasks due to which, in each task model, no task is scheduled for core3. In contrast, in hybrid scheduler, all the three cores are utilized as migration of tasks is allowed across cores and work load is distributed among the cores which can be seen in the result Gantt charts in Fig.6. The comparison of percentage of core utilization on three CPU cores for three task models is shown in Fig.7.

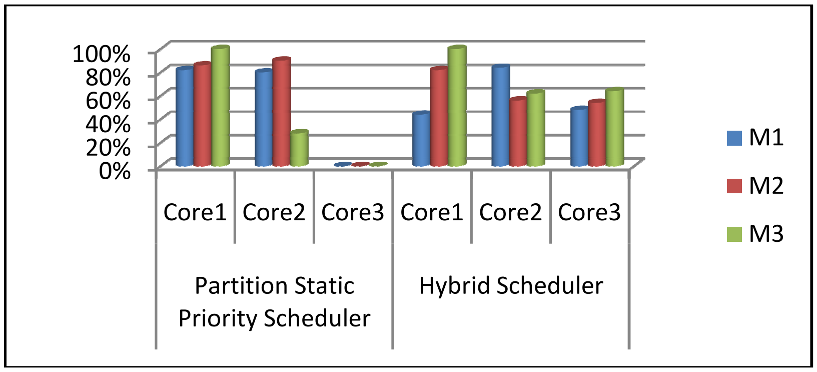

Fig.7 Comparison of percentage of core utilization on three models

7.2 Average response time

Average Response Time		
Task Models	Partition Static Priority Scheduler	Hybrid Scheduler
M1	5.1ms	3.2ms
M2	5.7ms	2.7ms
M3	3ms with 5 tasks scheduled	3.1ms

Table.2 Average Response Time

The response time for each scheduled task within the simulation duration is the time spent by a task from its release till completion that is:

Response time (Rt) = ET+WT, where ET= execution time and WT= waiting time

Average response time is the sum of response times of all scheduled tasks divided by the number of tasks scheduled that is:

$\text{Average response time} = (\sum_{i=1}^{n} Rt)/n$, Where n= no of scheduled tasks [15].

It can be observed from table 2 that, there is a considerable improvement in response time of tasks scheduled by hybrid algorithm over the partitioned algorithm for the task models 1 and 2. For task model 1, the improvement is of 1.9ms and for task model 2, it is of 3 ms. In task model 3, higher priorities are assigned to high frequency 5ms tasks. So partitioned static scheduler could not schedule for five low priority tasks due to which, response time is calculated only for five scheduled tasks. For this task model also there is an improvement in response time with the hybrid scheduler. The comparison of average response time of tasks for the three task models with both the algorithms is shown in Fig.8.

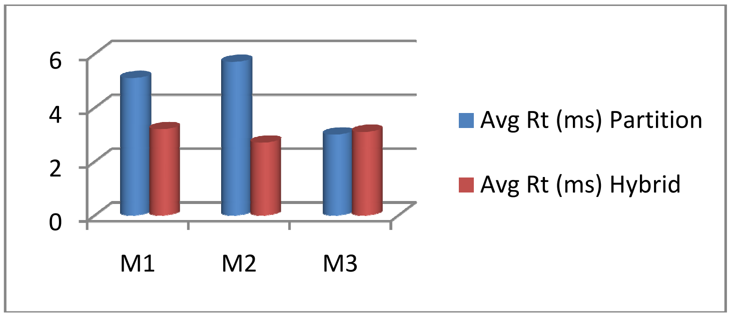

Fig.8 Average response time comparison

7.3 Task deadline missing rate

Deadline of a task is not assigned as a parameter in the task model. Since all the tasks considered are periodic, deadline of a task is assumed as equal to its period. For each of the scheduling algorithms, the number of tasks missing their deadlines on every task model within the simulation duration was observed. It was observed that, as the total utilization of the CPU core exceeds 60% for a task model, for partitioned static priority scheduling, lower priority tasks miss their deadlines or do not get scheduled. As the task model 3 has the utilization of 2.2, five of the tasks miss their deadlines, as a remedial measure for which, the average utilization of the tasks should be restricted to the range of $0.1 \leq [1/n(\sum Ci/Pi)] \leq 0.2$ where n is the number of tasks under test and 'i' varies from 1 to n. The number of tasks partitioned for a core should be restricted to have maximum 60 % work load. Experiments show that, hybrid scheduling algorithm provides a feasible schedule for the task models with 80% utilization of each core at any given time and deadline missing rate is 20-40% of the waiting task as the utilization goes beyond 80%.

7.4. Migration and Preemption Overheads

As task migration is allowed in hybrid scheduling algorithm either to meet the deadline or to utilize a comparatively less loaded core, it depends on the number of tasks get added to a local ready queue by the distribution logic and the slack of the task at the end of a queue after sorting in ascending order. Since these parameters are independent of each other, a definite number of task migrations cannot be derived. In this work, based on the task models, maximum number of task released at any instant is four. So maximum two tasks could be added to a queue and there could be total three tasks waiting in the queue. In this consequence, after comparison of slack at the end of the queue and looking at the load at other cores, maximum 30% of the tasks could migrate. Since these task migrations happen prior to execution time, preemption of tasks is reduced compared to the partitioned scheduling approach. As preemption incurs context switch overhead, cost of preemption overhead is always more than migration overhead. So hybrid scheduling algorithm implementing task migration logic has response time improvements over partitioned scheduling.

8.CONCLUSION

In this paper, a hybrid scheduling algorithm is proposed along with a scheduler model for multicore automotive ECUs. The existing algorithm used in this domain is the partition static priority scheduling wherein one core is utilized upto 60% of its capabilities and others remain idle in normal working condition of the vehicle. When load increases, it gets transferred to other cores. So two cores are underutilized in normal running conditions. In this paper, both the algorithms have been tested for three task models, each comprising of ten numbers of tasks representatives of Engine Control ECU functionalities. It has been verified that, this proposed algorithm has considerable improvements over the existing partitioned static priority scheduler based on the performance parameters such as: CPU core utilization, average response time of tasks and deadline missing rate. Each of these parameters is calculated based on theoretical knowledge. The main motive behind developing this hybrid scheduling algorithm was to distribute the tasks among the available cores instead of strict partitioning according to the task periods and executing at one core all through in normal working conditions of the vehicle. In the proposed algorithm, tasks were allowed to migrate from one queue to another, hence able to reduce the response time and meet the deadlines. As all the cores share the workload, higher utilization of the cores has been achieved when the work load increases. The algorithm is yet to be tested in different contingency conditions, could be considered as the future work.

9.REFERENCES

[1] ”Multicore Scheduling in Automotive ECUs” By Aurelien Monot, Nicolas Navet, Francoise Simonot, Bernard Bavoux, Embedded Real Time Software and Systems- ERTSS May 2010.

[2] AUTOSAR version 4.2.1, www.autosar.org

[3] “Parallel Real Time Scheduling of DAGs”, By Saifullah A; Ferry D; Jing Li; Agarwal K, Parallel and Distributed Systems, IEEE Transactions. Jan 2014.

[4] “Improvement of Real time Multicore Schedulability with forced Non Preemption.” By Jinkyu Lee; Shin K G , Parallel and Distributed Systems, IEEE Transactions. Jan 2014.

[5] “A High utilization Scheduling scheme of stream programs on clustered VLIW stream Architectures.” By Guoyue Jiang; Zhaolin Li; Fang Wang; Shaojun Wei, Parallel and Distributed Systems, IEEE Transactions, March 2013.

[6] “Dynamic Scheduling for Emergency Tasks on Distributed Imaging Satellites with Task Merging.” By Jianjiang Wang; Xiaomin Zhu; Dishan Qiu; Yang L T, Parallel and Distributed Systems, IEEE Transactions, June 2013.

[7] “Harmonic Aware Multicore Scheduling for fixed priority real time systems.” By Ming Fan; Geng Quan, Parallel and Distributed Systems, IEEE Transactions. March 2013

[8] “Understanding Multitask Schedulability in Duty-Cycling Sensor Networks.” By Mo li; Zhenjang Li; Longfei Shanguan; Shaojie Tang; et al, Parallel and Distributed Systems, IEEE Transactions. March 2013.

[9] ”A Case Study in Embedded Systems Design: An Engine Control Unit” , By Tullio Cuatto, Claudio Passerone, Claudio Sanso E, Francesco Gregoretti, Attila JuresKa, Alberto Sangiovanni, Design Automation for Embedded Systems, Vol, 6, 2000.

[10] “An Efficient Hierarchical Scheduling Framework for the Automotive Domain” By Mike Holenderski, Reinder J. Bril and Johan J. Lukkien, Real-Time Systems, Architecture, Scheduling, and Application, www.intechopen.com.

[11] N. Navet, A. Monot, B. Bavoux, and F. Simonot-Lion. “Multi-source and multicore automotive ecus: Os protection mechanisms and scheduling”. In Proc. of IEEE Int’l Symposium on Industrial Electronics, Jul. 2010.

[12] “Dynamic Task Scheduling on Multicore Automotive ECUs” By Geetishree Mishra, K S Gurumurthy, International Journal of VLSI design & Communication Systems (VLSICS) Vol.5, No.6, December 2014.
[13] “Hyperbolic Utilization Bounds for Rate Monotonic Scheduling on Homogeneous Multiprocessors.” By Hongya Wang; Lihchyun Shu; Wei Yin; Yingyuan Xiao, Jiao Cao ,Parallel and Distributed Systems, IEEE Transactions. August 2013
[14] “Real Time Scheduling Theory: A Historical Perspective” By Lui Sha, Tarek Abdelzaher, Karl Erik Arzen, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, Aloysios K.Mok; Real-Time Systems , Vol 28, PP-101-155, 2004.
[15] “Real-Time Scheduling Models: an Experimental Approach” By Antonio J Pessoa de Magalhaes, Carlos J.A. Costa, Technical report Controlo Nov 2000.
[16] ”Scheduling Hard Real-Time Systems: A Review” By A Burns- Software Engineering Journal, Vol 6, May1991, DOI: 10.1049/sej.1991.0015 IET.
[17] “Task Scheduling of Real-time Systems on Multi-Core Architectures” By Pengliu Tan, 2009 Second International Symposium on Electronic Commerce and Security IEEE, DOI 10.1109/ISECS.2009.161.
[18] ”Demand-based Schedulability Analysis for Real Time Multicore Scheduling” By Jinkyu Lee, Insik Shin, Journal of Systems and Software ELSEVIER October 2013.
[19] “An Experimental Comparison of Real Time Schedulers on Multicore Systems” By Juri lelli, Dario Faggioli, Tommaso Cusinotta, Giuseppe Lipari, Journal of Systems and Software ELSEVIER June 2012.
[20] “Load-prediction Scheduling Algorithm for Computer Simulation of Electrocardiogram in Hybrid Environments” By Wenfeng Shen, Zhaokai Luo, Daming Wei, Weimin Xu, Xin Zhu, Journal of Systems and Software ELSEVIER January 2015.
[21] “Performance Analysis of EDF Scheduling in a Multi priority Preemptive M/G/1 Queue” By Gamini Abhaya V; Tari Z; Zeeplongsekul P; Zomaya A Y , Parallel and Distributed Systems, IEEE Transactions. July 2013.