Journal of Engineering Science and Technology
Vol. 10, No.10 (2015) 1372 - 1393

A QUANTITATIVE STUDY BASED ON DIRECT MEASUREMENT ON EMBEDDED PROCESSORS LIMITING CONTEXT SIWTCHES FOR ENERGY SAVING

ANJU S. PILLAI*, T. B. ISHA

Department of Electrical and Electronics Engineering,
Amrita Vishwa Vidyapeetham, Coimbatore, Tamilnadu, India
*Corresponding Author: s_anju@cb.amrita.edu

Abstract

Priority based preemptive schedulers are preferred over non-preemptive schedulers due to their flexibility to accommodate real time tasks based on criticality. The overhead associated with a preemptive scheduler is high and with increased number of preemptions and the associated context switches, the execution pattern of tasks become highly unpredictable at run time. In this paper, an effort is made to provide an insight into the significance of controlling context switches during real time application development. The system under study consists of an ARM7 LPC2148 microcontroller, whose energy consumption measurement is carried out with the help of MBED NXP 1768 controller. A study is done by analytical verification and a software simulation using embedded C with Keil uVision IDE. The energy consumed by the processor with and without context switches is verified experimentally by direct measurement. The two factors considered for analysis are increased delay and the augmented energy dissipation during a context switch. It is seen that a substantial saving of time and energy is associated with every context switch.

Keywords: Context switch, Energy dissipation, Energy measurement, Simulation, Real time embedded application.

1. Introduction

Use of embedded systems is very prevalent in almost all application fields like bio-medical, engineering, process control, industrial and many more. The development of real time applications, especially real time embedded systems need more care and attention as it has to cater to critical nature. Scheduler assigns priorites to tasks based on their application nature and criticality to execute on the processor. In addition, need of a preemptive scheduler is also vital for successful

Nomenclatures

C_i	Execution time of task τ_i in μsec.
C_{a_eff}	Effective load capacitance in μF
D_i	Deadline of task τ_i in μsec.
E_i	Energy consumed by processor P_i in mJ
f_i	Clock frequency of processor P_i in MHz.
H	Hyper period
N_i	Number of clock cycles taken by task τ_i to execute
O_i	Operating point of processor P_i, which consists of supply voltage and clock frequency
P_i^j	The power dissipated by the processor while executing the j^{th} instance of the task τ_i in Watts (W)
T_i	Time period of task τ_i in μsec.
v_i	Supply voltage of processor P_i in Volts (V)

Greek Symbols

τ_i	Task *i*
τ_i^p	p^{th} instance of task τ_i

Abbreviations

DFS	Deterministic Stretch-to-Fit
DVS	Dynamic Voltage Scaling
FPP	Fixed Preemption Point
IDE	Integrated Development Environment
ISR	Interrupt Service Routine
LCM	Least Common Multiple
PTS	Preemption Threshold Scheduling
PLL	Phase Locked Loop
RTOS	Real Time Operating System
TCB	Task Control Block
WCET	Worst Case Execution Time

execution of various tasks in the system to guarantee functional and temporal requirements. Thus, use of a priority based preemptive scheduler becomes highly demanding for real time application development and scheduling of tasks. But, the decision of permitting preemptions must be made judiciously, so that the increased number of preemptions and the associated context switches are not introducing additional overhead to the system to compromise the temporal constraints or even losing schedulability of the system. When a preemption occurs in the system by stopping the execution of a low priority task by a more critical task, the context of the preempted task need to be saved into the memory and then, to continue the task execution, the RTOS kernel need to bring in and set up the environment of the new incoming task. This activity known as context switch is one of the most costly operation of any RTOS. Thus, the context switches in the system need to be controlled to retain only the necessary ones, to facilitate the successful execution of both critical and non-critical tasks.

Plenty of research works are carried out in the field of real time systems pertaining to control of context switches, still there are avenues for further

exploration. Finding an optimal solution for overheads associated with context switches and analysing its effects are challenging. There are many adverse effects of context switches which include: increased delay for task execution, increased memory requirements, cache associated delays and unpredictable task execution patterns, manipulating task queues and many more [1-3]. Cache related preemptions are costly and it is found that the worst case delay can go as high as 655μsec. These delays when gets added up to the execution time of the task, it is found that an increase of around 33% of execution time was observed in a MPC7410 Power PC with a two way set associative L2 cache [1]. Thus, to use a preemptive scheduler in an optimal manner, the overhead associated with context switches has to be bounded. Otherwise the advantage of having more flexibility and better processor utilization can be taken away by the non-preemptive scheduler.

In the present work, an analysis is made to present the severity of context switch overheads viz. increased context switch delay and added energy dissipation of the processor. In this work, a prototype model consisting of ARM7 LPC2148 and MBED NXP 1768 controller is considered and the energy consumed by LPC2148 controller while executing tasks with and without context switches is measured by actual measurement. This is a small step made in the direction of actual measurement of the core level performance of an embedded processor.

The rest of the paper is organized as follows: section 2 contains a brief description about the related work done in the field and sections 3 emphasises on the system model. In section 4, methodology and solution are presented followed by results and discussion in section 5. Finally section 6 concludes the paper.

2. Related Work

The process of context switch is very essential in any application field especially in real time scheduling. Context switches help to incorporate tasks with more critical and stringent temporal requirements to the processor for successful execution. There is a wide spread acceptance for the need for controlling context switches in the literature. The major overheads which are mentioned earlier includes the increased time delay, energy consumption, memory requirements, cache related overheads, unpredictable task execution patterns, and it may even lead to infeasibility to schedule the tasks [4]. There are many different approaches which are suggested by researchers in the field to overcome the various costs. DVS is a key technique for energy consumption reduction by switching the voltage and frequency of the processor at runtime to slowdown/increase the speed of the task execution. DVS can be used to control context switches by regulating the speed of task execution [5, 6]. In [7], algorithmic power management schemes are presented to limit the power dissipation of the embedded processor which in turn is beneficial in controlling the context switching overheads. Another well-known technique which is widely used in the field is PTS [8]. In PTS technique, every task poses a threshold value in addition to the task priorities. In this system, a task can preempt another task only when its priority is greater than the threshold level of the other task. Another class of method uses slack reclamation technique, wherein the unused processor time is utilized to slow down task instances to conserve energy and reduce preemptions. In [9], Baruah investigated the possibilities of preemption limiting and made an attempt to find the longest possible non-preemptive execution time for sporadic tasks in a system. In [10],

the authors proposed methods to fix preemption points in task execution, assuming a fixed preemption cost. Bhatti et al. [11] proposes DSF method to control preemptions and inter task and intra task mechanism by Koedam are some techniques for effective context reduction [12]. Kim and Jim introduced accelerated completion based technique and delayed preemption based control technique for limiting preemptions [13] and a job phasing aware preemption deferral was proposed by Marinho [14]. The preemption control was also attempted by varying the tasks parameters in [15]. In [16], a detailed analysis of limited preemptive scheduling algorithms is presented and has showed that FPP algorithm is one of the algorithms that produce less number of preemptions with increased schedulability ratio. But it demands inclusion of explicit preemption points in the program code. In [17] fixed priority scheduling with deferred preemption is proposed which guarantees the performance and defines a compromise between the response time bound and processor utilization.

Energy saving could be done by compiler optimization techniques by appropriately selecting power aware instructions. In this case, there is no need of a hardware modification. When compared to hardware based techniques, simulation approach is less complex, but there is a drawback of not possible to account for factors which affect the power consumption viz. heat loss, external atmospheric conditions and many more [18]. In the literature, there are a few research works which support for power consumption estimation through hardware based methods. The amount of power consumed can be measured by measuring the voltage and current flowing through a resistor with the aid of milli-ampere meter. This method tries to eliminate the costly off-chip memory accesses by storing the future data for computations by saving into registers [19, 20]. In [21] power consumption measurement of Intel PXA255 computing module is carried out by measuring the voltage drop across the resistor which is connected to the external power supply and by calculating the instantaneous current through the circuit. The DVS technique is implemented on ARM clusters using the available monitoring systems and an average of 20% saving was observed [22]. Further by incorporating more power saving techniques like extended sleep mode, an extra 15% more saving can be obtained [23, 24]. A comprehensive review of distinct methods which can be employed for controlling context switches can be found in [25].

This paper proposes a prototype model which can be employed for measurement of power consumption in a processor core level. The actual current and voltage measurement of the embedded processor while executing a task code can be measured by this prototype model. At present, the focus of measurement is time delay and energy consumption associated with a context switch.

3. System Model

This section describes the system model. The task model, execution time model, scheduling model and energy model are presented in the subsequent subsection.

3.1. Task model

A set of periodic tasks $\Gamma = \{\tau_1, \tau_2, \ldots . \tau_n\}$ are considered for the analysis. Each periodic task is depicted by the following attributes: execution time C_i, time period T_i and deadline D_i. The execution time of each task is assumed as equal to its WCET. The WCET of the task is computed when tasks are run on the processor

operated at rated conditions, i.e., at rated voltage v_{rated} and rated clock frequency f_{rated}. In the current model tasks which are independent in nature are considered, which means that tasks do not share any common resources. Also, task execution is considered not to suspend any task till the execution is completed.

3.2. Execution time model

The focus of the current work is to investigate the impact of context switch on energy consumption. Theoretical concepts are developed to analyze the variation in energy consumption per context switch when the processor operating points are changed. Operating point of the processor is defined by a set of voltage and frequency values. Investigations are carried out to find preferable operating points of embedded processor to reduce the energy dissipation per context switch, ensuring both functional and temporal constraints. For every processor, there is a range of possible operating conditions, which are defined between (OP_{low} to OP_{high}), where: OP_{low} defines the lower possible voltage and clock frequency which is required by the processor to execute and deliver the normal functionalities, and OP_{high} defines the maximum supply voltage and clock frequency at which the processor could be operated.

3.3. Processor model

The processor model described consists of an embedded processor which operates at discrete points. The discrete operating points $O = \{o_1, o_2, \ldots . o_p\}$, are couple of voltage and frequency values defined within the permissible ranges, i.e., $o_m = \{v_m, f_m\}$. Each v_m value belong to the class of voltage ranges defined within the minimum and maximum voltage, i.e., $v_m \in \{v_{min}\ to\ v_{max}\}$ and $f_m \in \{f_{min}\ to\ f_{max}\}$. Thus, a DVS enabled processor is required for switching the operating point of the embedded processor/controller. Operating the processor at a lower operating point is always desirable to conserve energy and switching to a lower operating point is permitted only after ensuring temporal requirement of the system.

In the current work, an ARM7 LPC2148 microcontroller is considered. The LPC2148 works with a supply voltage of 2.9-3.3V and frequencies in the range of 12 MHz-60 MHz. The different operating points of LPC2148 are shown in Table 1.

Table 1. Operating points supported by ARM7 LPC2148 microcontroller.

Operating Points	Frequency (MHz.)	Voltage (V)
$\mathbf{O_1}$	12	2.9
$\mathbf{O_2}$	24	3.0
$\mathbf{O_3}$	36	3.1
$\mathbf{O_4}$	48	3.2
$\mathbf{O_5}$	60	3.3

3.4. Energy model

The energy model defines the different parameters on which the energy consumption of an embedded processor depends. When the processor speed is

varied by varying the supply voltage $v = v_1, v_2, \ldots \ldots v_p$ and clock frequency $f = f_1, f_2, \ldots \ldots f_p$, there is a change in the execution pattern of the tasks. This variation may sometimes be desirable as it ensures temporal requirements with a slight increase in energy, or compromises the performance with the reward of decreased energy consumption. The total energy consumed by the energy aware embedded processor for the hyper period length is computed as [26]:

$$E_{total} = \sum_{i=1}^{n} \sum_{j=1}^{\frac{H}{T_i}} P_i^j * C_i^j \quad (1)$$

where H is the LCM of task time periods. In the above expression, P_i^j represent the power dissipated by the processor while executing the j[th] instance of the task τ_i and C_i^j is the time required by the task code to execute on the processor, i.e., task execution time. Thus, energy consumption is the rate of change of power dissipation. The power dissipation of the processor is estimated by Eq. (2).

$$P_i^j = C_{a\_eff} \sum_{i=1}^{n} \left\{ \sum_{j=1}^{\frac{H}{T_i}} v_i^{j^2} * f_i^j \right\} \quad (2)$$

where v_i^j is the supply voltage at which the processor is operated and f_i^j is the clock frequency of the processor while executing the j[th] instance of task τ_i. C_{a_eff} is the effective load capacitance of the processor.

4. Methodology

In the current work, an attempt to quantify the overheads of context switch viz. time delay and energy consumption by actual measurement is carried out. The methodology consists of three different approaches for evaluating the energy dissipation associated with a context switch:

- Analytical computation of energy dissipation associated with a context switch
- A direct measurement to experimentally quantify the time delay associated with a context switch, current and voltage consumed by the ARM controller while executing real time tasks to compute the energy consumption with and without context switches for task execution when run on an ARM7 controller
- Simulation approach to measure the time delay and energy wastage associated with context switch

Details of these three approaches are elucidated in the following subsections and the complete methodology steps are described in the flowchart shown in Fig. 1.

4.1. Analytical evaluation

Execution of two task instances τ_i^p and τ_j^q, can be done either in a non-preemptive manner in which task with high priority is executed and completed followed by the

low priority task, or in preemptive scheduler where in the release of a critical task will always preempt the currently executing low priority task.

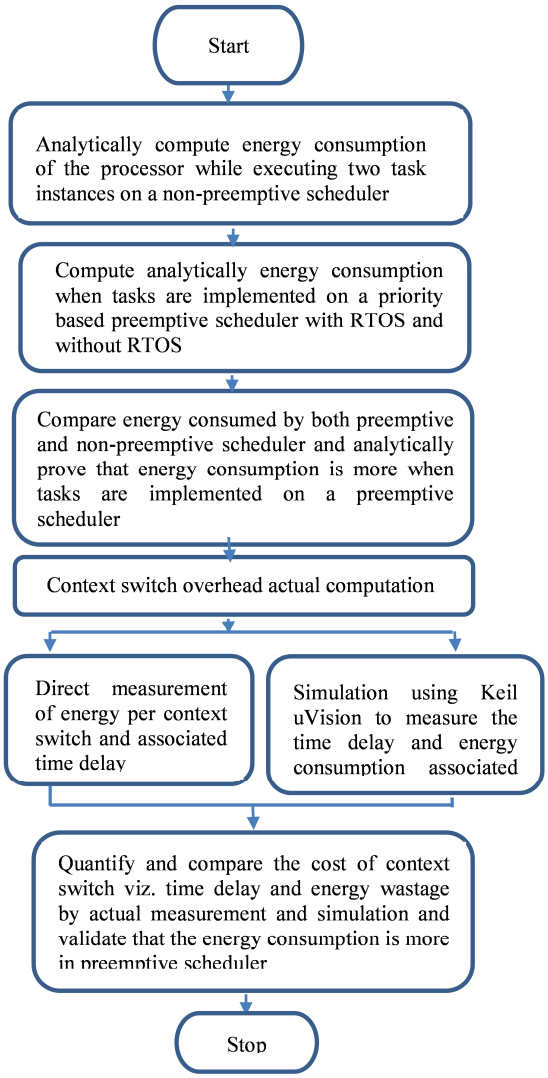

Fig. 1. Steps involved in proposed work.

4.1.1. Calculation of energy consumption when tasks are executed on a non-preemptive scheduler

Under a non-preemptive scheduler, the overheads associated with context switch are not present for the task execution. By knowing the operating points of the embedded processor, the energy dissipated for executing the task instances on a processor can be estimated. Let the processor under consideration be P_i, operated

with a supply voltage of v_{rated} and clock frequency of f_{rated}. Also, by knowing the processor on which the tasks are executed (in the present model, an ARM7 LPC2148 microcontroller); the time required to execute an instruction can be computed. This is done by knowing the number of clock cycles required to run an instruction. Thus, for the written real time task code, cumulatively adding the number of clock cycles for each instruction will give the total number of clock cycles taken by that processor to execute the given real time task. The execution pattern for the tasks in a non-preemptive scheduler is described in Fig.2.

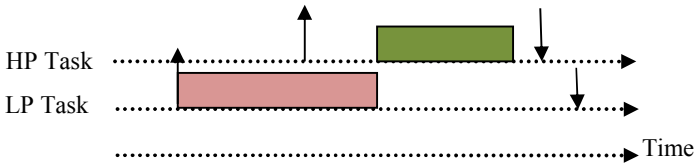

Fig. 2. Task execution on priority based non-preemptive scheduler.

Lemma 4.1.1.1 The energy consumption of an embedded processor while executing two task instances $\boldsymbol{\tau_i^p}$ and $\boldsymbol{\tau_j^q}$ on a non-preemptive scheduler can be computed as:

$$E_{total\_wo} = C_{a\_eff} * v_{rated}^2 * \left\{\sum_{k=1}^{l} N_k + \sum_{k=1}^{m} N_k\right\} \quad (3)$$

Proof. Let the number of clock cycles is represented as $\boldsymbol{N_{i\_wo}}$ for executing the real time task $\boldsymbol{\tau_i}$ on a processor $\boldsymbol{P_i}$. Where $\boldsymbol{N_{i\_wo}}$ is given as:

$$N_{i\_wo} = \sum_{k=1}^{l} N_k$$

where, $\boldsymbol{N_k}$ is the number of clock cycles required to run an instruction of task $\boldsymbol{\tau_i}$. Similarly, the number of clock cycles required to run the high priority real time task is given by:

$$N_{j\_wo} = \sum_{k=1}^{m} N_k$$

Thus, the energy consumed by processor P_i with a non-preemptive scheduler (E_{i_wo}) can be calculated as in equation 4.

$$E_{i\_wo} = Power\ dissipation * Time\ \ taken\ to\ execute\ the\ task\ \tau_i^p \quad (4)$$

$$= \{C_{a\_eff} * v_{rated}^2 * f_{rated}\} * \left\{No.\ of\ clock\ cycles * \frac{1}{f_{rated}}\right\}$$

$$= C_{a\_eff} * v_{rated}^2 * N_{i\_wo}$$

$$= C_{a\_eff} * v_{rated}^2 * \sum_{k=1}^{l} N_k \quad (5)$$

Similarly,

$$E_{j\_wo} = C_{a\_eff} * v_{rated}^2 * \sum_{k=1}^{m} N_k \quad (6)$$

The above expression implies that as the code length increases with increase in clock cycles, the energy consumption also increases. The operating voltage also has an impact on energy consumption, which is quadratic in nature.

The total energy consumption of the embedded processor to execute two task instances $\boldsymbol{\tau_i^p}$ and $\boldsymbol{\tau_j^q}$ on a non-preemptive scheduler is:

$$E_{total\_wo} = E_{i\_wo} + E_{j\_wo} \quad (7)$$

$$E_{total\_wo} = C_{a\_eff} * v_{rated}^2 * \left\{ \sum_{k=1}^{l} N_k + \sum_{k=1}^{m} N_k \right\} \quad (8)$$

4.1.2. Calculation of energy consumption when tasks are executed on a preemptive scheduler

When tasks τ_i^p and τ_j^q are executed on a priority based preemptive scheduler, tasks are assigned to the processor for execution based on their priority. Such schedulers allow task preemptions to facilitate a quick response for critical tasks. Any real time application implementation could be done with or without using RTOS. Anyway, there are some overheads associated in both cases.

a) Task Implementation with RTOS

When tasks are implemented using RTOS, RTOS assign a specific memory space to the task and bring the code to be executed by the task into that specific memory. The RTOS then instantiate a data structure called TCB. The RTOS use TCB block to store all information pertaining to the task in order to handle and schedule the task. The information contained in the TCB include the task ID, task state, starting address of the task code, temporary register contents, program counter, status register etc. The context of a task thus refers to different data and register contents that define the condition of a task. When tasks are implemented on RTOS kernel with a priority based preemptive scheduler, the RTOS need to perform a set of activities for every context switch. The scheduler need to find:

1. Whether the task under execution should continue to run on the processor
2. The next immediate task to be executed
3. Saving context of the preempted task
4. Set up the surroundings for the next task to run, and
5. Allow the new task to run

The above actions will happen simultaneously at the time of a context switch. While responding to task preemption, every RTOS need a definite time to handle

it. At the time of context switching, the RTOS need to save the context of the preempted task and all register values into the memory, and take up the context of the new incoming task. After completion of execution of the high priority task, the RTOS again need to bring in the saved context of the preempted task into the memory for resuming task execution. There is a time delay associated with this. This delay is the time required by the RTOS to handle an interrupt. In the analysis these two latencies are considered separately. When context switch occurs, an ISR is called to run the high priority task (δ_{t1})and at the completion of ISR, the RTOS kernel is activated to switch to the low priority task execution (δ_{t2}). This execution scenario is shown in Fig. 3.

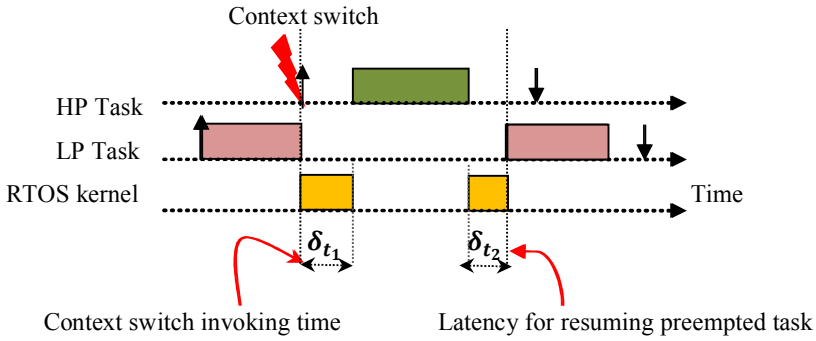

Fig. 3. Task execution on a priority based preemptive scheduler.

As the number of context switches increases for a task, these additional context switch latencies will get added up to its execution time, which may some time lead to task deadline miss.

Theorem 4.1.2.1 *The amount of energy consumption when tasks are implemented on a priority based preemptive scheduler with RTOS is greater than that of a non-preemptive scheduler.*

Proof. The different attributes of the low priority task $\boldsymbol{\tau_i^p}$ are: $(\boldsymbol{C_i, D_i, T_i})$ and the respective values of the high priority task $\boldsymbol{\tau_j^q}$ are $\boldsymbol{C_j, D_j, T_j}$. When tasks are implemented on a preemptive scheduler with RTOS, the execution time of the task is modified due to the associated preemption and context switch overheads. The new execution time of the low priority task is:

$$C_{i\_new} = C_i + C_{\delta t2} \tag{9}$$

where $C_{\delta t2}$ is the latency for resuming preempted task execution. The overhead of context switch namely: writing the context of the preempted task into the memory and retrieving the content back from the stack is split and added into both preempting task and preempted task respectively. Thus, the new execution time of the high priority task is:

$$C_{j\_new} = C_j + C_{\delta t1} \tag{10}$$

where $\boldsymbol{C_{\delta t1}}$ is the latency associated with context switch invoking time. Therefore, the number of clock cycles of low priority and high priority task code now becomes:

$$N_{i\_w\_rtos} = \sum_{k=1}^{l+\delta t2} N_k \quad \text{and} \quad N_{j\_w\_rtos} = \sum_{k=1}^{m+\delta t1} N_k$$

Thus, the energy consumed to implement the task instance τ_i^p is:

$$\begin{aligned} E_{i\_w\_rtos} &= Power\ dissipation * Time\ taken\ to\ execute\ task\ \tau_i^p \\ &= \left\{C_{a\_eff} * v_{rated}^2 * f_{rated}\right\} \left\{No.of\ clock\ cycles * \frac{1}{f_{rated}}\right\} \\ &= \left\{C_{a\_eff} * v_{rated}^2 * f_{rated}\right\} \left\{N_{i\_w\_rtos} * \frac{1}{f_{rated}}\right\} \\ &= \left\{C_{a\_eff} * v_{rated}^2\right\} \left\{\sum_{k=1}^{l+\delta t2} N_k\right\} \end{aligned} \tag{11}$$

And that of task τ_j^q is:

$$E_{j\_w\_rtos} = \left\{C_{a\_eff} * v_{rated}^2\right\} \left\{\sum_{k=1}^{m+\delta t1} N_k\right\} \tag{12}$$

The total energy consumed while executing the above two tasks are obtained by adding the individual task consumption.

$$E_{total\_w\_rtos} = E_{i\_w\_rtos} + E_{j\_w\_rtos} \tag{13}$$

$$\begin{aligned} &= C_{a\_eff} * v_{rated}^2 \left\{\sum_{k=1}^{l+\delta t2} N_k + \sum_{k=1}^{m+\delta t1} N_k\right\} \\ &= C_{a\_eff} * v_{rated}^2 \\ &\quad * \left\{\left\{\sum_{k=1}^{l} N_k + \sum_{k=1}^{m} N_k\right\} + \left\{\sum_{k=1}^{\delta t1} N_k + \sum_{k=1}^{\delta t2} N_k\right\}\right\} \end{aligned} \tag{14}$$

By using Lemma 4.1.1.1, the energy consumption of an embedded processor while executing two task instances in a non-preemptive scheduler is found as:

$$E_{total\_wo} = C_{a\_eff} * v_{rated}^2 * \left\{\sum_{k=1}^{l} N_k + \sum_{k=1}^{m} N_k\right\}$$

Therefore, $E_{total\_w\_rtos}$ can be written as:

$$E_{total\_w\_rtos} = E_{total\_wo} + C_{a\_eff} * v_{rated}^2 * \left\{ \sum_{k=1}^{\delta t1} N_k + \sum_{k=1}^{\delta t2} N_k \right\} \quad (15)$$

Thus, it is proved that the amount of energy consumption when tasks are implemented on a priority based preemptive scheduler with RTOS is greater than that of a non-preemptive scheduler.

b) Task Implementation without RTOS

For small application development, there is no need to use an RTOS, as the programmer can manage the task interactions by skillful coding. Under such cases, the coded program should be capable of checking the task priorities at the release of every new task into the system, and the scheduler needs to take a decision of whether the current task under execution or the newly released task should run. Therefore, the scheduling decisions are made at the release of every task instances. Thus, the coded program must include these decision making instructions and additional instructions to resume the task execution of preempted task on completion of the preempting task. In case of multiple preemptions, the program should be written well to handle and keep track of nested preemptions suffered by a task. Thus, additional instructions must be written in the program so as to handle these context switch overheads. The task execution without RTOS is presented in Fig. 4.

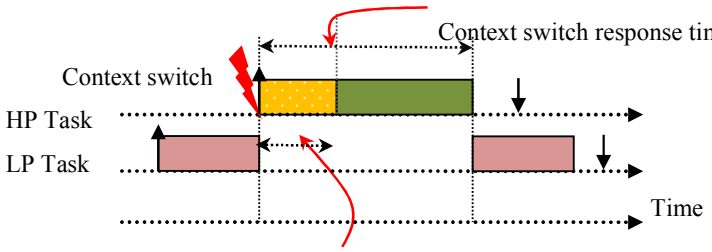

Fig. 4. Task execution without RTOS.

Theorem 4.1.2.2 *The amount of energy consumption when tasks are implemented on a priority based preemptive scheduler is greater than that of a non-preemptive scheduler.*

The new execution time of low priority task instance τ_i^p and high priority task instance τ_j^q can be written by including an additional time $C_{\Delta t\_i}$ required to run the extra instructions given in equation 16.

$$C_{i\_new} = C_i + C_{\Delta t\_i} \quad (16)$$

$$C_{j\_new} = C_j + C_{\Delta t\_j} \quad (17)$$

And the corresponding number of clock cycles required to run the tasks τ_i^p and τ_j^q are:

$$N_{i\_w} = \sum_{k=1}^{l+\Delta t\_i} N_k$$

$$N_{j\_w} = \sum_{k=1}^{m+\Delta t\_j} N_k$$

The energy consumption of the processor while executing the tasks can be written as:

$$E_{i\_w} = \left\{C_{a\_eff} * v_{rated}^2 * f_{rated}\right\} \left\{N_{i\_w} * \frac{1}{f_{rated}}\right\}$$

$$= \left\{C_{a\_eff} * v_{rated}^2\right\} * \sum_{k=1}^{l+\Delta t_i} N_k \quad (18)$$

And that of task τ_j^q is:

$$E_{j\_w} = \left\{C_{a\_eff} * v_{rated}^2\right\} * \sum_{k=1}^{m+\Delta t\_j} N_k \quad (19)$$

The total energy consumption of the embedded processor while executing two task instances on a priority based preemptive scheduler without RTOS is given by:

$$E_{total\_w} = E_{i\_w} + E_{j\_w}$$

$$= C_{a\_eff} * v_{rated}^2 \left\{\sum_{k=1}^{m+\Delta t\_i} N_k + \sum_{k=1}^{l+\Delta t\_j} N_k\right\}$$

$$= C_{a\_eff} * v_{rated}^2 * \left\{\left\{\sum_{k=1}^{l} N_k + \sum_{k=1}^{m} N_k\right\} + \left\{\sum_{k=1}^{\Delta t\_i} N_k + \sum_{k=1}^{\Delta t\_j} N_k\right\}\right\} \quad (20)$$

By using Lemma 4.1.1.1, the energy consumption of an embedded processor while executing two task instances in a non-preemptive scheduler is found as:

$$E_{total\_wo} = C_{a\_eff} * v_{rated}^2 * \left\{\sum_{k=1}^{l} N_k + \sum_{k=1}^{m} N_k\right\} \quad (21)$$

Therefore, E_{total_w} can be written as:

$$E_{total\_w} = E_{total\_wo} + C_{a\_eff} * v_{rated}^2 * \left\{ \sum_{k=1}^{\Delta t\_i} N_k + \sum_{k=1}^{\Delta t\_j} N_k \right\} \quad (22)$$

Thus, it is proved that the amount of energy consumption when tasks are implemented on a priority based preemptive scheduler is greater than that of a non-preemptive scheduler, with the support of RTOS and without RTOS.

4.2. Study by direct measurement

This section details the experimental rig for measurement and the procedure used to assess the time required to perform a context switch during two task execution and the voltage and current consumed by the embedded processor/controller while implementing task execution and context switch. The measurement system shown in Fig. 5 consists of an ARM7 TDMI based LPC2148 microcontroller where the execution of real time task is performed. The real time tasks which are considered for evaluation are an UART transmission task and Multiply Accumulate (MAC) code. The measuring setup consists of the test processor (LPC2148 microcontroller) on which the task execution and context switches are implemented, and an MBED NXP1768 mbed microcontroller to compute the instantaneous power dissipation of LPC 2148 controller. The analog voltage and current values of LPC 2148 controller are digitized and fed to MBED NXP1768 to compute the instantaneous power dissipation of LPC2148 when the tasks are executed with and without context switches. The current transducer used is an AE make (0-100mA) range meter which gives an equivalent output voltage in the range of (0-3.2V). A Moving Coil (M.C.) voltmeter (0-5V) is used for voltage measurement.

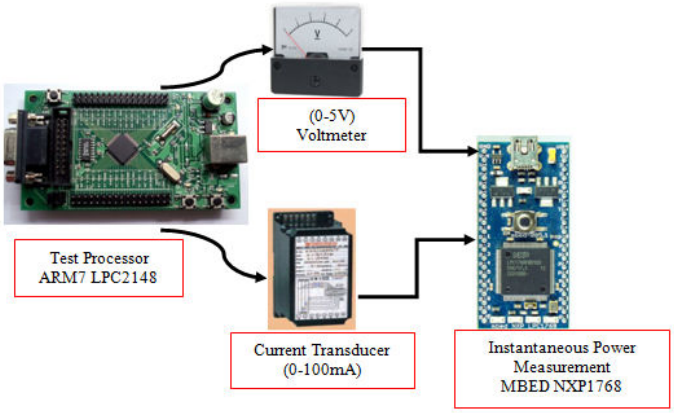

Fig. 5. Measuring setup.

4.2.1. Measuring the time required for a context switch

The time required to perform a context switch by LPC2148 microcontroller, when operated at the rated conditions of 3.3V and 60 MHz crystal frequency are

obtained experimentally. For this, a task code to implement UART transmission is written and the number of clock cycles required to complete the code was computed by using Timer 1 module of LPC2148. Upon the completion of UART transmission code, the value of the Timer 1 module is displayed to the hyper terminal. The second task which is considered is a MAC program to multiply numbers and add the products to find the result. The MAC code is run for different number of iterations to verify the changes in number of clock cycles required during execution. Both task codes are written in Embedded C and the code after successful compilation and building are flashed into LPC microcontroller using Keil- uVision IDE.

Two separate programs are written one without context switch, i.e., sequential execution of UART transmission program followed by the MAC computation. Another program is written to implement context switch by executing the UART transmission task first and preempting its execution by the MAC task and upon its completion, the UART task resumes its execution for culmination. Timer 1 is configured and initialized during the starting of program execution without context switches and Timer 2 for program with context switches. The difference of values of Timer 2 and Timer 1 gives the number of clock cycles required to carry out context switch.

4.2.2. Measuring the energy consumption for a context switch

The supply voltage and current consumed by LPC2148 microcontroller, while executing the program without context switch and with context switch are measured. By knowing the time taken for execution of both the programs, the energy consumed in each case is estimated as below:

- Measure the time taken by the LPC to execute the two identified tasks UART and MAC operation in non-preemptive manner and with preemptions as explained in section 4.2.1 and let the time measured be $t_{with-pre}$ and $t_{without-pre}$.
- The supply voltage of LPC microcontroller is fixed at the rated value of 3.3V and clock frequency of 60 MHz.
- Thus the energy consumed by the LPC microcontroller while running the tasks with and without preemptions is calculated as in Eq. 23 and 24.

$$E_{without-pre} = C_{a\_eff} * v^2 * f * t_{without-pre} \quad (23)$$

$$E_{with-pre} = C_{a\_eff} * v^2 * f * t_{with-pre} \quad (24)$$

The energy dissipated by the controller when a context switch is implemented is the difference between the energy consumption with and without preemption as below:

$$E_{cont-switch} = E_{with-pre} - E_{without-pre} \quad (25)$$

Figure 6 shows a snapshot of the output of MBED NXP1768 controller while computing the instantaneous energy consumption of LPC2148 microcontroller while implementing context switch.

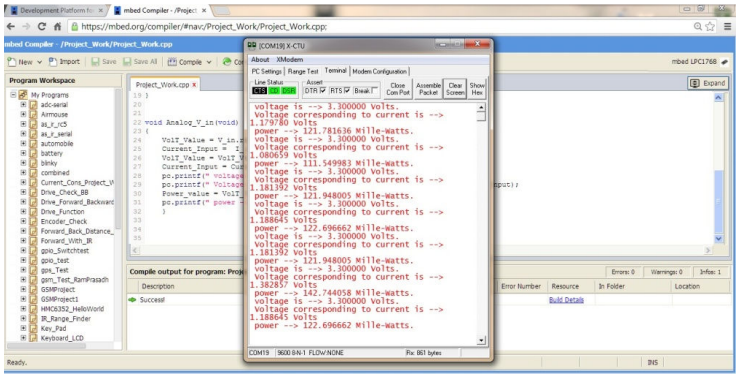

Fig. 6. MBED NXP1768 controller to compute energy consumption.

4.3. Simulation using Keil uVision IDE

The context switch implementation using software is carried out using Keil-uVision IDE. Here, the two programs written for task execution with and without context switches using embedded C language are compiled using Keil IDE. ARM7 LPC2148 microcontroller is programmed through Flash Magic tool. In Flash Magic tool appropriate settings of COM port selection, baud rate, device selection and clock frequency is to be done to set the appropriate values as in the experiment. Using the start/stop debug session, get the disassembly listing and then single stepping operation is performed. By taking the execution profile and enabling the *show time* option, the time taken to execute each instruction of the code is displayed. By adding up individual instruction execution time, the time taken to complete the entire task code is calculated. The difference of the time obtained by running the program with context switch and without context switch is a measure of the time consumed to implement the context switch.

5. Results and Discussion

In this work, the time delay and the associated energy wastage of an embedded processor while implementing context switches are quantified by actual measurement. In ARM7 LPC2148 microcontroller, the real time tasks namely, UART transmission and MAC functional programs are implemented with and without context switches. The voltage input and current consumed by LPC microcontroller are measured with the help of a M.C. voltmeter and current transducer. These analog values are digitized and input to MBED NXP1768 controller, which computes the instantaneous energy dissipation of LPC controller while executing tasks. The experimental setup used for time delay measurement and energy consumption is shown in Fig. 7 and a close view of the components in Fig. 8.

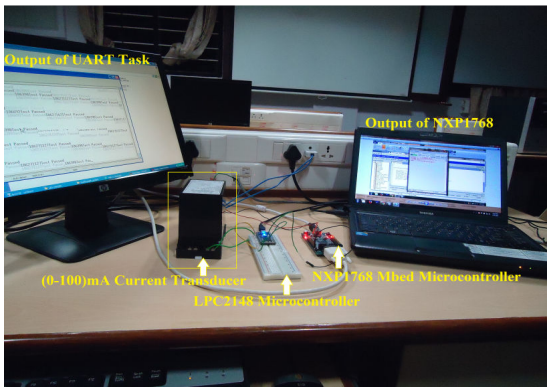

Fig.7. Experimental setup for measurement of time delay and energy consumption.

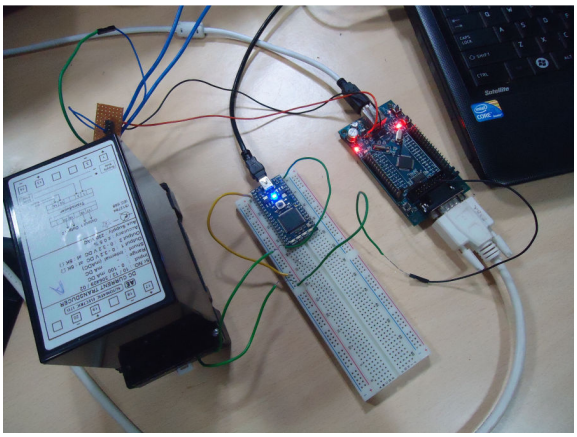

Fig. 8. A close view of components.

When the UART transmission task and MAC program with different number of iterations are executed with and without context switches, the number of clock cycles taken by the LPC2148 microcontroller is tabulated in Table 2. These readings are taken when LPC2148 is operated at rated voltage of 3.3V and frequency of 60 MHz.

Figure 9 shows a graph between the context switching time for the LPC2148 microcontroller when operated at rated voltage of 3.3V and at varying frequencies of 12 MHz, 24 MHz, 36 MHz, 48 MHz and 60 MHz. The oscillator is intended for calibration in multiples of the base frequency and this fixes the selection of frequency switches as 12, 24, 36, 48 and 60 MHz. The variations in clock frequency are achieved by configuring PLL and the change in voltage is accomplished with a digital potentiometer connected across the power supply and the test system. MCP4021 is used which has 64 step changes. Here, the MAC program is executed with 4 iterations.

Table 2. Number of clock cycles and time measurement of ARM7 LPC2148 microcontroller to implement context switch.

	MAC with 4 operations(No. of clock cycles)		**MAC with 16 operations(No. of clock cycles)**		**MAC with 32 Operations (No. of clock cycles)**	
	With Context Switch	Without Context Switch	With Context Switch	Without Context Switch	With Context Switch	Without Context Switch
UART Transmission	186565	186302	193757	193244	214515	213642
No. of clock cycles for context switch	263		513		873	
Time taken for a context switch (µsec.)	4.38		8.55		14.55	

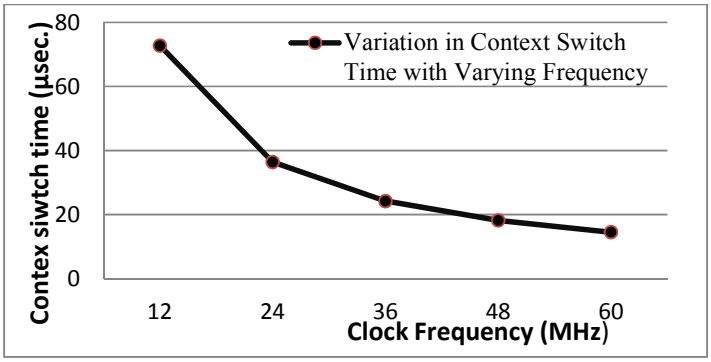

Fig. 9. Time required for context switching with varying frequencies.

For the measurement of energy consumption per context switch, the supply voltage and current consumption of LPC2148 microcontroller is measured while executing the UART and MAC task codes with and without context switch. Fig. 10 shows the variation in energy consumption with varying number of MAC operations at varying operating points. The different operating points used are as given in Table 1.

By simulation, when UART and MAC code are run with different number of operations, the values obtained are tabulated in Table 3. The LPC2148 is run at rated operating conditions of 3.3V and 60 MHz clock frequency.

Upon computing the time delay associated per context switch while executing two tasks: UART transmission and MAC operation, by direct measurement and by simulation, the results obtained are presented in Fig. 11, when the controller is run at rated operating point of 3.3V and 60 MHz. Also, Fig. 12 presents the comparison of energy consumption by the above two approaches.

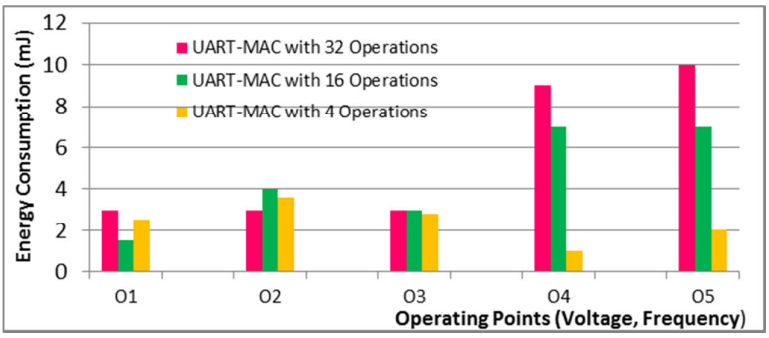

Fig. 10. Variation in energy consumption with change in operating points.

Table 3. Number of clock cycles and time measurement of ARM7 LPC2148 microcontroller using software simulation.

	UART & MAC with 4 operations	UART & MAC with 16 operations	UART & MAC with 32 operations
Time taken for a context switch (µsec.)	4.16	8.43	14.28

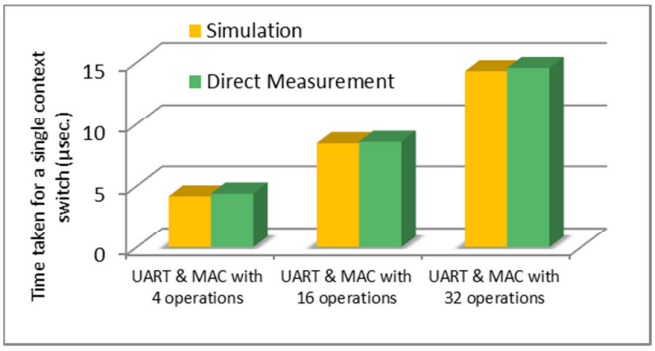

Fig. 11. Comparison of time delay by simulation and actual measurement.

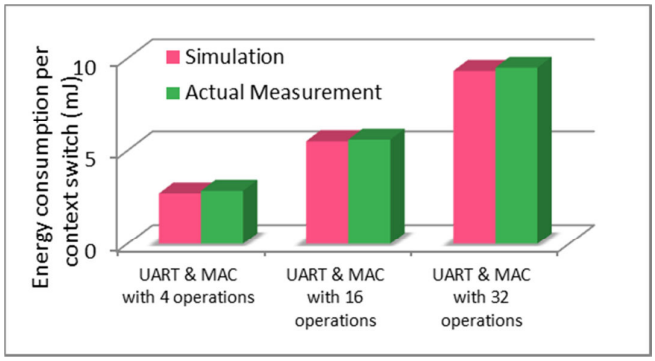

Fig. 12. Comparison of energy consumption per context switch by simulation and actual measurement.

An important observation made is that the time for context switching and energy consumption is not the same while running different tasks, i.e., with change in application tasks, there is a change in time required for context switching, thereby changing the energy consumption. Also, another consideration is, the context switching time and energy dissipation changes with change in context switching point. Based on the application nature, the preferable operating points can be selected. Running the processor at lower voltage and frequency are always preferable to reduce energy consumption if temporal requirements of the application tasks are not violated.

6. Conclusion

For any real time application development consisting of many concurrent tasks, the use of a priority based preemptive scheduler becomes inevitable for every task to meet their respective functional and temporal requirements. Under such scenario, when developing big and complex real time applications, it becomes necessary to have large number of context switches in the system. But, if not controlled, the unnecessary context switches may incur lot of overheads including increased delay, increased memory, cache pollution and sometimes even infeasibility of the system. This paper attempts to provide insight into the significance of context switches in real time embedded application development and render the impact of context switch in application development. The main focus of the work is to quantify the amount of delay and energy wastage associated with context switches while different application tasks are executed. The main two factors which are investigated is the time delay associated with a context switch and the energy consumption per context switch. In this paper, analytical proofs are presented to verify the fact of increase in time delay and energy consumption during a context switch by comparing the task execution with and without context switches. Experiments are conducted to quantify these factors on ARM7 LPC2148 microcontroller and MBED NXP1768 controller. The experimental results show a substantial amount of time delay and energy wastage associated with context switches while running simple real time tasks. The energy consumption in the order of few milli Joules cannot be neglected as the cumulative value for a large number of context switches will be in the order of Joules. Thus, the number of context switches in a system need to be limited for energy saving. Also, the context switch behaviour and the associated overhead for branch instructions are not simple and straight forward as with simple instructions presented in the current work. The associated overhead prediction and analysis need to be investigated in the future work.

References

1. B. D. Bui, M. Caccamo, L. Sha, and J. Martinez (2008). Impact of cache partitioning on multi-tasking real time embedded systems, in The *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications.*101-110.
2. H. Ramaprasad and F. Mueller (2008). Tightening the bounds on feasible preemptions, in *The ACM Transactions on Embedded Computing Systems.* 212-224.

3. C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim (1998). Analysis of cache-related preemption delay in fixed-priority preemptive scheduling, *The IEEE Transactions on Computers*. 47(6), 700-713.
4. G. Yao, G. Buttazzo, and M. Bertogna (2010). Comparative evaluation of limited preemptive methods, in *The International Conference on Emerging Technologies and Factory Automation.*1-8.
5. A. Thekkilakattil, A. S. Pillai, R. Dobrin, and S. Punnekkat (2010). Reducing the number of preemptions in real-time systems scheduling by CPU frequency scaling, in *The International Conference on Real-Time and Network Systems*.
6. Jejurikar and R. K. Gupta (2004). Integrating processor slowdown and preemption threshold scheduling for energy efficiency in real time embedded systems, in *The IEEE Real-Time Computing Systems and Applications.*
7. Macro E.T.Gerards (2014). Algorithmic power management: Energy minimization under real-time constraints, Ph.D.Thesis Series No. 14-314, Centre for Telematics and Information Technology, University of Twenty, The Netherlands.
8. Y. Wang and M. Saksena, (1999). Scheduling fixed-priority tasks with preemption threshold, in *The International Conference on Real-Time Computing Systems and Applications*. 328-335.
9. S. Baruah, (2005). The limited-preemption uniprocessor scheduling of sporadic task systems, in The Euromicro Conference on Real-Time Systems, 137-144.
10. M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo (2010). Preemption points placement for sporadic task sets, in *The Euromicro Conference on Real-Time Systems.*251-260.
11. Bhatti, M.K.; Belleudy, C.; Auguin, M. (2010). An inter-task real time DVFS scheme for multiprocessor embedded systems, *Design and Architectures for Signal and Image Processing (DASIP), Conference on*, 136-143, 26-28 Oct.
12. Koedam, M, Stuijk, S, Corporaal, H. (2011). Exploiting Inter and Intra Application Dynamism to Save Energy, *Digital System Design (DSD), 14th Euromicro Conference on*, 708-715.
13. Woonseok Kim; Jihong Kim; Sang Lyul Min (2004). Preemption-Aware Dynamic Voltage Scaling in Hard Real-Time Systems, *Low Power Electronics and Design, ISLPED '04. Proceedings of the 2004 International Symposium on*, 393-398.
14. J. Marinho, S. Petters (2011). Job phasing aware preemption deferral, *9th International Conference on Embedded and Ubiquitous Computing (EUC)*, IFIP, 128–135. doi:10.1109/EUC.2011.46.
15. R. Dobrin, G. Fohler (2004). Reducing the number of preemptions in fixed priority scheduling, *Proceedings 16th Euromicro Conference on Real-Time Systems*, ECRTS, 144 – 152. doi:10.1109/EMRTS.2004.1311016.
16. Buttazzo, G.C.; Bertogna, M.; Gang Yao (2013). Limited Preemptive Scheduling for Real-Time Systems. A Survey, *Industrial Informatics, IEEE Transactions on*, 9(1), 3-15.

17. T. H. C. Nguyen, N. S. Tran, V. H. Le, and Pascal Richard (2013). Approximation scheme for real-time tasks under fixed-priority scheduling with deferred preemption. In *Proceedings of the 21st International conference on Real-Time Networks and Systems* (RTNS '13). ACM, New York, NY, USA, 265-274.
18. Tajana Simuni c, Luca Benini, Giovanni De Micheli, Mat Hans (2000). Source Code Optimization and Profiling for Energy Consumption in Embedded Systems, *IEEE Symposium on System Synthesis,* 193 –198.
19. Jeffry T. Russell, Margarida F Jacome (1999). Software Power Estimation and Optimization for High Performance 32-bit Embedded Processors, *IEEE Proceedings of ICCD'98,*328-333, 5-7 Oct.
20. Theodore Laopoulos, Periklis Neofotistos, C. A. Kosmatopoulos, and Spiridon Nikolaidis (2003). Measurement of Current Variations for the Estimation of Software Related Power Consumption, *IEEE Transactions on Instrumentation and Measurement,* 52(4), 1206-1212.
21. Lin, Jian, Cheng, Albert, Song, Wei (2014). A Practical Framework to Study Low-Power Scheduling Algorithms on Real-Time and Embedded Systems, *J. Low Power Electron. Appl.* 2014, *4*(2), 90-109.
22. Zhonghong Ou, Bo Pang, Yang Deng, Jukka K. Nurminen, Antti Yla- Jaaski, Pan Hui (2012). Energy and Cost-Efficiency Analysis of ARM-Based Clusters, *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid),* 115-123.
23. Ravindra Jejurikar, Rajesh Gupta (2004). Dynamic Voltage Scaling for System-wide Energy Minimization in Real-Time Embedded Systems, *Proceedings of the 2004 International Symposium on Low Power Electronics and Design,* 78 – 81.
24. P. Pillai, K. G. Shin (2001). Real-time dynamic voltage scaling for low power embedded operating systems, *IEEE Symposium on Operating Systems Principles,*89–102.
25. G. Buttazzo, M. Bertogna, and G. Yao (2012). Limited preemptive scheduling for real-time systems: A survey, *The IEEE Transactions on Industrial Informatics*, 9(1), 3-15.
26. Neil H.E. Weste, K. Eshraghian, Principles of CMOS VLSI Design, A Systems Persoectives, (2nd ed.). *Person Education*, India.