Applied Mathematical Modelling 38 (2014) 1929–1947

Contents lists available at ScienceDirect

Applied Mathematical Modelling

journal homepage: www.elsevier.com/locate/apm

A Monte Carlo algorithm for real time task scheduling on multi-core processors with software controlled dynamic voltage scaling

Abhishek Mishra [a,*], Anil Kumar Tripathi [b]

[a] Center of Excellence in Information & Communication Technology, Indian Institute of Technology Jodhpur, Old Residency Road, Ratanada, Jodhpur 342 011, India
[b] Department of Computer Engineering, Indian Institute of Technology (Banaras Hindu University), Varanasi 221 005, India

ARTICLE INFO

Article history:
Received 28 March 2012
Received in revised form 17 July 2013
Accepted 8 October 2013
Available online 25 October 2013

Keywords:
Dynamic voltage scaling
Energy efficient scheduling
Multi-core processor
Randomized algorithm

ABSTRACT

The task scheduling problem for multi-core processors is an important algorithm design issue. Dynamic voltage scaling (DVS) is used to reduce the energy consumption of cores. We ponder the problem of task scheduling on a multi-core processor with software controlled DVS where the objective is to reduce the energy consumption. We consider a system with a single multi-core processor with software controlled DVS having a finite set of core speeds and discuss a task scheduling problem associated with it. The problem that we address is to find a minimum energy task schedule for a given set of independent tasks that have to be completed within a given common deadline. We propose a Monte Carlo algorithm of complexity $O(t(mp+q+log(t))+p(t+q)(D^{pq}+n))$ for solving the task scheduling problem and compare it with the optimal algorithm. Here t is the number of tasks, p is the number of cores, q is the number of core speeds, m is an integer parameter that is the number of iterations we should try to get a feasible solution before declaring that no solution is possible, n is an integer parameter that is the number of iterations we should try to reduce the energy consumption when we get a feasible solution, and D is the common deadline of the tasks.

1. Introduction

In order to meet the challenges of high performance computing applications, a good solution is to use the multi-core processor architecture [1,2]. A multi-core processor is composed of two or more independent cores on a single chip. Multi-core processors are used in a variety of applications such as general purpose, embedded, network, digital signal processing (DSP), graphics etc.

Dynamic voltage scaling (DVS) is a power management technique in which varying the supply voltage can vary the speed of cores [3]. We can reduce the voltage (undervolting) to save the energy when computational requirement is low. We can also increase the voltage (overvolting) to improve the system performance when computational requirement is high.

This gives rise to the task scheduling problem [4] for multi-core processors in which the objective is to find a schedule of core speeds/voltages and also a schedule of tasks so as to minimize the energy consumption given a set of independent tasks to process and a deadline.

* Corresponding author. Tel.: +91 291 2449032.
E-mail addresses: amishra@iitj.ac.in, abhishek.rs.cse@itbhu.ac.in (A. Mishra), aktripathi.cse@itbhu.ac.in (A.K. Tripathi).

0307-904X/$ - see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.apm.2013.10.023

We consider systems with a single multi-core processor with software controlled DVS that has a finite set of core speeds. We address the problem of energy efficient task scheduling. The problem is to find a minimum energy task scheduling for a given set of independent tasks that have to be completed within a given common deadline. For solving this problem we propose a *Monte Carlo* algorithm of complexity $O(t(mp+q+log(t))+p(t+q)(D^{pq}+n))$ and compare it with the *optimal (OPT)* algorithm. Here t is the number of tasks, p is the number of cores, q is the number of core speeds, m is an integer parameter that is the number of iterations we should try to get a feasible solution before declaring that no solution is possible, n is an integer parameter that is the number of iterations we should try to reduce the energy consumption when we get a feasible solution, and D is the common deadline of the tasks.

The rest of the paper is organized as follows. Section 2 presents the literature overview. In Section 3 we consider systems with a single multi-core processor. In Section 4 we give some motivating examples. In Section 5 we address the *Energy Efficient Task Scheduling Problem (EETSP)*. In Section 6 we propose a Monte Carlo algorithm (the ENERGY-EFFICIENT-TASK-SCHEDULING-ALGORITHM - *EETSA*) for solving the EETSP problem. In Section 7 we compare the EETSA algorithm with the OPT (optimal) algorithm for dual-core processors. In Section 8 we compare the EETSA algorithm with the OPT algorithm for quad-core processors. Finally we conclude in Section 9.

2. Literature overview

There are a number of energy efficient task scheduling algorithms proposed in the literature for a wide range of system models. There are some examples of uniprocessor energy efficient algorithms [5–11]. There are some examples of multiprocessor energy efficient scheduling algorithms [12–18,4]. Some algorithms assume a continuously varying processor speed [6,4,17]. While some others assume a discretely available processor speeds [8,6,7,10,19].

Ishihara and Yasuura [8] consider the problem of voltage scheduling on a uniprocessor with DVS having a small number of discretely variable voltages. In their result they show that the voltage-scheduling problem of minimizing the energy consumption by a processor for processing a given computational load with a given deadline can be solved optimally in polynomial time with at most two voltages.

Yao et al. [10] solve the problem of minimum energy scheduling of independent jobs with arrival times, deadlines, and a given amount of computation on a uniprocessor with variable speeds. They assume that the power function is a convex function of the processor speed. They consider the case of discretely available processor speeds. They give an $O(nlog^2(n))$ time optimal offline algorithm for the problem where n is the number of jobs. Their result is extended by Irani et al. [19] to include the case in which a processor can go into a sleep state. In the sleep state, the processor speed and its power consumption is 0, but a constant amount of energy is required to bring back the processor into a non-sleep mode. They propose a 3-approximation algorithm for the offline version of the problem.

Chen et al. [7] extend the problem of Yao et al. [10] to include the case of jobs with precedence constraints. They consider the case of weakly dynamic voltage scheduling in which speed change is not allowed in the middle of processing a job. They prove the problem to be NP-Complete and also give fully polynomial-time approximation schemes for some special cases of the problem.

Yang et al. [4] consider the problem of energy efficient scheduling for a chip-multiprocessor with DVS that can use continuously varying processor speeds with no upper bound. The power function is assumed to be cubic in processor speed. They consider the problem of voltage scheduling for a set of independent tasks that share a common deadline. They give a 2.371-approximation algorithm for the problem.

Zhang et al. [17] consider the problem of energy efficient scheduling of real time dependent tasks on a given number of variable voltage processors. They give a two-phase framework to solve the problem. The first phase is for task assignment and ordering. The second phase is the voltage selection (VS) phase. They formulate the VS problem as an integer-programming (IP) problem. They prove that the IP problem for VS can be solved in polynomial time for the case of processors with continuous voltages.

3. Systems with a single multi-core processor

We are considering systems with a single multi-core processor with software controlled DVS having discretely available core speeds. Some examples of software controlled DVS are *Enhanced Intel SpeedStep Technology* [20], and *AMD PowerNow! technology* [21]. For example, *Enhanced Intel SpeedStep Technology* [20] for the Intel Pentium M processor supports processor speeds of 600 MHz to 1.6 GHz with a step of 200 MHz. For having low software overhead in switching the voltages, we assume that the DVS software is a periodic process that wakes up periodically to check if there is a need to change the voltage, and otherwise it sleeps.

We assume the power consumption function of the core to be cubic in the core speed [4]:

$$P(s)=\alpha s^3, \tag{1}$$

where α is a constant and s is speed of the core.

The energy consumed by a core running at a speed of s for τ time units is given by $P(s)\tau$. We assume that the overheads in changing the supply voltages are negligible. We also assume that any core can be taken into a sleep mode with $s = 0$, but all of non-sleeping cores must run at the same speed. The computational work done c in cycles of a core running at a speed of s for τ time units is assumed to be given by:

$$c = s\tau. \tag{2}$$

We assume that there are p homogeneous cores in the multi-core processor. We also assume that the DVS software is a periodic process so that the supply voltage can change only in steps of a certain amount of time ($= \delta\tau$). Without loss of generality we take this time step as our unit of time ($\delta\tau = 1$). There are q possible core speeds (non-zero) that are given by:

$$Q = \{s_i | 1 \leqslant i \leqslant q, s_i \in \mathbb{N}\}. \tag{3}$$

4. Motivating examples

Scheduling of independent tasks with a common deadline is an important real-time task scheduling problem. Such type of problems arise in frame-based systems in which all tasks start at time 0 and should be finished before the end of the frame [22].

A frame-based system is described by Liu et al. [23] in which they describe an industrial application that collects data from thousands of sensors at several MHz rate. We can have an independent task assigned for each sensor that collects the data from the sensor and then processes it for further analysis or visualization. All tasks should be finished within their respective time frames in order that no data are missed.

In military applications involving fighter aircrafts, warships or submarines, various sensors send their data periodically to a central computer on which the data are processed for displaying the targets. For example in submarines we have radar, sonar, periscope, ESM etc. [24] that independently track their respective targets. The various sensors can send their data periodically to a central computer on which we can assign an independent task for each sensor that collects the data from its corresponding sensor, processes it, and displays it appropriately. All tasks should be finished within the their corresponding time frames so that no important data are missed.

Consider the video surveillance applications [25] that simultaneously display many real time videos on a single screen. We can assign an independent task for each video that receives the picture frame from its corresponding video, resizes it, and maps it on the screen. All tasks should be finished within their corresponding time frames so that no picture frames are missed.

In the study of migration patterns of migratory birds or animals by using GPS tracking devices [26], birds or animals are tagged with GPS tracking devices that periodically send the GPS coordinates. We can have an independent task assigned for each GPS tracking device that receives the GPS coordinate from its corresponding sensor, processes the data, and displays it on a graphical map. All tasks should be finished within their corresponding time frames so that no coordinates are missed.

5. The *Energy Efficient Task Scheduling Problem (EETSP)*

Let $\mathbb{N}$ denote the set of natural numbers. For $a \in \mathbb{N}$ and $b \in \mathbb{N}$ with $a \leqslant b$, let $[a..b]$ denote the set of natural numbers between a and b including a and b. Matrices are assumed to be integer matrices having each component as an integer. An $m \times n$ matrix $V_{m\times n}$ is written as $V_{m\times n} = (v_{ij})_{m\times n}$ where (i,j) is used to index the components of $V_{m\times n}$ as v_{ij} where $(i,j) \in [1..m] \times [1..n]$. We denote the zero matrix of dimension $m \times n$ having all entries as 0 by $O_{m\times n}$. A $1 \times n$ matrix $U_{1\times n}$ is written as $U_{1\times n} = (u_j)_n$ where j is used to index the components of $U_{1\times n}$ as u_j where $j \in [1..n]$. A matrix $W_{m\times n}$ having each row as $U_{1\times n}$ is written as: $W_{m\times n} = (U)_{m\times 1}$. By the inequalities involving matrices, we mean the set of inequalities involving the corresponding elements of the matrices.

Let there be t independent tasks given by:

$$T = \{T_j | j \in [1..t]\}, \tag{4}$$

having computational requirements given by the matrix $C_{1\times t}$:

$$C_{1\times t} = (c_j)_t, \quad c_j \in \mathbb{N}, \tag{5}$$

where the task T_j requires c_j amount of computation for $j \in [1..t]$. The tasks have to be scheduled on a multi-core processor with p cores. The set of possible core speeds (non-zero) is given by the set Q (Eq. (3)). The common deadline of tasks is $D \in \mathbb{N}$. A task can only be executed on a single core non-preemptively.

Let the speed profile for time in $[0, D]$ be given by:

$$F_{1\times q} = (f_i)_q, \quad f_i \in \mathbb{N} \cup \{0\}, \tag{6}$$

where f_i is the number of time slots in $[0, D]$ that run at the speed of s_i for $i \in [1..q]$. We have the following speed profile constraint:

$$\sum_{i=1}^{q} f_i \leqslant D. \tag{7}$$

The maximum work done by a core using this speed profile is given by:

$$M = \sum_{i=1}^{q} s_i f_i. \tag{8}$$

We have to partition the set T into disjoint subsets

$$G = \{G_k | k \in [1..p]\}, \tag{9}$$

such that for each G_k, for $k \in [1..p]$, we have:

$$\sum_{j=1.T_j \in G_k}^{t} c_j \leqslant M. \tag{10}$$

Let $G_{p\times t}$ be the binary partition matrix of the set T such that:

$$g_{kj} \in \{0, 1\}, \quad (k, j) \in [1..p] \times [1..t], \tag{11}$$

$$\sum_{k=1}^{p} g_{kj} = 1, \quad j \in [1..t], \tag{12}$$

$$\sum_{j=1}^{t} c_j g_{kj} \leqslant M, \quad k \in [1..p]. \tag{13}$$

Let $H_{p\times q}$ be the sleep matrix. For $(k, i) \in [1..p] \times [1..q]$, h_{ki} is the number of time slots for s_i on the kth core that are sleeping. We have the following constraints for the sleep matrix:

$$h_{ki} \leqslant f_i, \quad (k, i) \in [1..p] \times [1..q]. \tag{14}$$

We have the work constraints given by:

$$\sum_{i=1}^{q} s_i (f_i - h_{ki}) \geqslant \sum_{j=1}^{t} c_j g_{kj}, \quad k \in [1..p]. \tag{15}$$

The energy consumed is given by:

$$E = \alpha \sum_{k=1}^{p} \sum_{i=1}^{q} s_i^3 (f_i - h_{ki}). \tag{16}$$

Definition 1. Given the input $(p, Q, C_{1\times t}, D)$, the *Energy Efficient Task Scheduling Problem (EETSP)* is to find $F_{1\times q}, G_{p\times t}$, and $H_{p\times q}$ such that (16) is minimized while also satisfying the constraints (6)–(8), (11)–(15).

Mishra and Tripathi [27] have proved the EETSP problem to be NP-Complete. They have also proved the EETSP problem to be inapproximable under the assumption of $P \neq NP$. That is the reason we have to propose a Monte Carlo algorithm (EETSA) in Section 6.

6. A randomized algorithm for EETSP

In this section we propose a *Monte Carlo* algorithm [28] for EETSP. The Energy-Efficient-Task-Scheduling-Algorithm (EETSA) is the proposed algorithm:

Algorithm 1. ENERGY-EFFICIENT-TASK-SCHEDULING-ALGORITHM (m, n, t, C, Q, D).

```
01 SORT (C)
02 (A, F, H) ← (O_{1×p}, O_{1×q}, O_{p×q})
03 f_q ← D
04 for l ← 1 to m do
05   flag ← TRUE
06   for j ← 1 to t do
07     min ← Ds_q − a_1
08     for k ← 1 to p do
09     d ← Ds_q − a_k
10     if (d ⩾ 0) ∧ (d < min) then
11       u ← k
12       min ← d
13     if min ⩾ 0 then
14       for k ← 1 to p do
15         g_kj ← 0
16       g_uj ← 1
17       a_u ← a_u + c_j
18     else
19       flag ← FALSE
20       break
21   if flag = FALSE then
22     A ← O_{1×p}
23     RANDOM-SWAP (1, j)
24   else
25     break
26 if flag = FALSE then
27   return NO SOLUTION
28 else
29   b ← ⌈t/n⌉
30   E′ ← α Σ_{k=1}^{p} Σ_{i=1}^{q} s_i^3 (f_i − h_ki)
31   (A, F′, G′, H′) ← (O_{1×p}, F, G, H)
32   for j ← 1 to t do
33     for k ← 1 to p do
34     a_k ← a_k + c_j g_kj
35   for r ← 1 to n do
36     F″ ← F
37     for v ← 1 to b do
38     RANDOM-SELECT (1, D)
39     M ← Σ_{i=1}^{q} s_i f_i
40     E ← α Σ_{k=1}^{p} Σ_{i=1}^{q} s_i^3 (f_i − h_ki)
41     if ∧_{k=1}^{p} (a_k ⩽ M) ∧ (E ⩽ E′) then
42     (E′, F′, G′, H′) ← (E, F, G, H)
43     else
44     F ← F″
45   for all H ∈ {H|(H ⩾ O_{p×q}) ∧ (H ⩽ (F)_{p×1})} do
46     if ∧_{k=1}^{p} (Σ_{i=1}^{q} s_i (f_i − h_ki) ⩾ a_k) then
47     E ← α Σ_{k=1}^{p} Σ_{i=1}^{q} s_i^3 (f_i − h_ki)
48     if E ⩽ E′ then
49       (E′, F′, G′, H′) ← (E, F, G, H)
50   return (E′, F′, G′, H′)
```

The inputs to EETSA are $m,\ n,\ t,\ C,\ Q$, and D. As defined in Section 5, t is the number of tasks, $C_{1\times t}$ is the computation time of tasks, Q is the set of possible core speeds (non-zero), and D is the given deadline. Here m is an integer parameter that is the number of iterations we should try to get a feasible solution before declaring that no solution is possible. Also n is an integer parameter that is the number of iterations we should try to reduce the energy consumption when we get a feasible solution.

The EETSA algorithm can be divided into two parts: the first part from lines 01 to 27; and the second part from lines 28 to 50. In the first part from lines 01 to 27, we try to find a feasible solution. In the second part from lines 28 to 50, we try to reduce the energy consumption when a feasible solution is found in the first part.

In line 01 we sort the tasks in non-increasing order of their execution times using the function SORT. In line 02 we initialize the matrices $A,\ F$, and H with the zero matrices of appropriate dimensions respectively. As defined in Section 5, $F_{1\times q}$ is the speed-profile, and $H_{p\times q}$ is the sleep-profile. $A_{1\times p}=(a_k)_p$ is used to calculate the amount of computation done on the cores. In line 03, to get a feasible solution we start with all cores running at the maximum speed. In the **for** loop from lines 04 to 25 we make m attempts to find a feasible solution. The variable *flag* in line 05 is initialized to TRUE which is used to find whether a feasible solution exists or not. If we are not able to find a feasible solution, then we return *NO SOLUTION* in the **if** block from lines 26 to 27. In the **for** loop from lines 06 to 20 we take the tasks in non-increasing order of their execution times and try to allocate them on the cores using the *worst fit* strategy [29,30]. The variable *min* is initialized in line 07 that is used to calculate the minimum remaining amount of computation by following the *worst fit* strategy. In the **for** loop from lines 08 to 12 we find the *worst fit* allocation for the jth task. In line 09 we store the remaining amount of computation in the variable d. In the **if** block from lines 10 to 12 we store the location in the variable u (line 11) and the value in the variable *min* (line 12) if a lower value of remaining amount of computation is found. The value of the variable *min* determines whether a task can be allocated or not (the **if-else** block from lines 13 to 20). If the allocation is not possible then we set the value of the variable *flag* to FALSE (line 19) in the **else** block from lines 18 to 20 and also break (line 20) out of the **for** loop from lines 06 to 20. Otherwise we set the corresponding values in the binary partition matrix G (as defined in Section 5) in the **for** loop from lines 14 to 15, and the line 16, and also update the amount of computation done on the core allocated in line 17. In the **if-else** block from lines 21 to 25, if we get a feasible solution (the **else** block from lines 24 to 25) then we proceed further to the second part of the algorithm (lines 28 to 50). Otherwise (the **if** block from lines 21 to 23) we initialize the matrix A with the zero matrix in line 22 and use the function RANDOM-SWAP in line 23 to randomly select two consecutive tasks between the first and the jth task and swap them so that we can try other random combinations to get a feasible solution in the next iteration of the **for** loop from lines 04 to 25.

In the **else** block from lines 28 to 50, initially we have a feasible solution corresponding to the cores running at the highest speed. In line 29 we set the value of b that is the maximum number of randomly selected time slots in which the core speeds are reduced in order to reduce the energy consumption. In line 30 we evaluate the energy consumption and store it in the variable E' that is used to record the minimum energy consumption. The matrices F', G', and H' corresponding to E' are initialized, and also the matrix A is set to the zero matrix in line 31. In the **for** loop from lines 32 to 34 the matrix A is set to the amount of computation done on each core. In the **for** loop from lines 35 to 44 we make n attempts to reduce the energy consumption. In line 36 we store the current value of F in F''. In the **for** loop from lines 37 to 38 we randomly select up to b time slots for speed reduction of cores using the function RANDOM-SELECT (line 38). The function RANDOM-SELECT has two randomization steps. The first randomization step randomly selects a time slot between the first and the Dth time slot. The second randomization step changes the value of F corresponding to the selected time slot. The new speed selected is randomly chosen to be smaller (non-zero) than the current speed at the selected time slot. In line 39 we evaluate the maximum possible work done M as defined in Section 5. In line 40 we calculate the energy consumption corresponding to the new value of F and store it in the variable E. If the energy consumption is reduced and the constraint (12) is satisfied, then we store the parameters corresponding to the smaller energy consumption in the **if** block from lines 41 to 42, otherwise we keep the previous value of F in the **else** block from lines 43 to 44 so that we can try other random combinations in the next iteration of the **for** loop from lines 35 to 44.

The energy consumption calculated so far corresponds to the value of the sleep-sleep matrix H being zero, in which case no core is sleeping. Now we try to further minimize the energy consumption by allowing the cores to sleep by changing the sleep-matrix H in the **for all** loop from lines 45 to 49. In the **for all** loop from lines 45 to 49 we select all possible values of H that are satisfying the constraint (13). In the **if** block from lines 46 to 49 we check the constraint (14) using the values of H. If the constraints are satisfied, then we evaluate the energy consumption in line 47. If this is reducing the energy, then we store the corresponding parameters in the **if** block from lines 48 to 49. Finally we return the solution in line 50.

Line 01 has complexity $O(tlog(t))$ [31]. Line 02 has complexity $O(pq)$. Line 03 has complexity $O(1)$. Considering the **for** loop from lines 04 to 25, the three levels of **for** loops (starting from lines 04, 06, and 08 respectively) contribute a complexity of $O(mtp)$. The **if-else** block from lines 13 to 20 contributes a complexity of $O(mtp)$ because of the **for** loop from lines 14 to 15, and because it is inside the two levels of **for** loops starting from lines 04, and 06 respectively. Line 22 has complexity $O(p)$. RANDOM-SWAP has complexity $O(1)$ (line 23). Therefore the **if-else** block from lines 21 to 25 contributes a complexity of $O(mp)$ because it is inside the **for** loop starting from line 04. Therefore the complexity of the **for** loop from lines 04 to 25 comes out to be $O(mtp)$. Line 29 has complexity $O(1)$. Line 30 has complexity $O(pq)$. Line 31 has complexity $O(pq+pt)$. The **for** loop from lines 32 to 34 has complexity $O(tp)$. RANDOM-SELECT has complexity $O(q)$ (line 38) because we have to search for the appropriate speed that corresponds to the time slot selected. Therefore the complexity of the **for** loop from lines 37 to 38

has complexity $O(bq) = O(tq/n)$ (from line 29). Lines 36 and 39 have complexity $O(q)$. Line 40 has complexity $O(pq)$. The **if-else** block from lines 41 to 44 has complexity $O(pq + pt)$. Therefore the **for** loop from lines 35 to 44 has complexity $O(npq + npt + tq)$. The complexity of the **if** block from lines 48 to 49 has complexity $O(pq + pt)$. Line 47 has complexity $O(pq)$. Therefore the **if** block from lines 46 to 49 has complexity $O(pq + pt)$. We can have a maximum of D^{pq} values of H in the **for all** loop starting from line 45. Therefore the **for all** loop from lines 45 to 49 has a complexity of $O(D^{pq}(pq + pt))$. Line 50 has a complexity of $O(pq + pt)$. By adding the above complexities we get the complexity of the EETSA algorithm as $O(t(mp + q + log(t)) + p(t + q)(D^{pq} + n))$.

Theorem 1. *When the algorithm EETSA returns a solution, then the solution satisfies the constraints 6, 7, 8, 10, 11, 12, 13, 14 as required in Definition 1.*

Proof. Constraints (6) and (7) are satisfied by lines 02 and 03. The only place where the value of F is changed is by the function RANDOM-SELECT in line 38. RANDOM-SELECT changes the value of F so that the constraints (6) and (7) are satisfied. Constraint (8) is satisfied by line 39. Constraints (10) and (11) are satisfied by the **for** loop from lines 14 to 15, and by the line 16. Constraint (12) is satisfied by the **if-else** block from lines 41 to 44. Constraint (13) is satisfied by line 02, and by the **for all** loop from lines 45 to 49. Constraint (14) is satisfied by the **if** block from lines 46 to 49. Therefore when the algorithm EETSA returns a solution, then the solution satisfies the constraints (6)–(8), (10)–(14) as required in Definition 1. □

7. Experimental results for dual-core processors

In this section we compare the EETSA algorithm with the optimal (OPT) algorithm. The OPT algorithm enumerates all possible solutions that satisfy all the constraints and evaluates the energy consumption. It gives the solution with the least energy consumption if a feasible solution exists. The OPT algorithm has exponential complexity. For the dual-core processor we take $\alpha = 1$, and $p = q = 2$. For the EETSA algorithm we take $m = n = 10$. The EETSA algorithm is run 100 times, and in the results we give the average and the standard deviation of energy consumed and the running time.

7.1. Symmetrical and asymmetrical task sets

By symmetrical task set we mean a task set that is having identical tasks as its task set. A system with symmetrical task set is the video surveillance application [25] (Fig. 1).

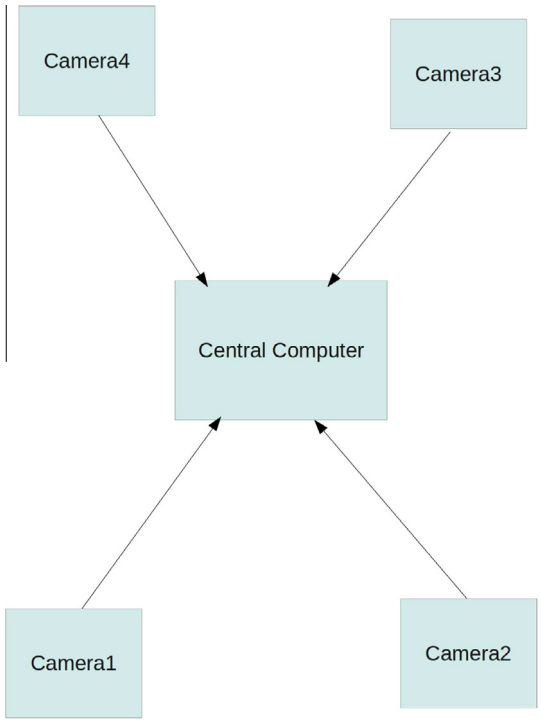

Fig. 1. Video surveillance system: a system with symmetrical task set.

By asymmetrical task set we mean a task set that may have non-identical tasks as its task set. A system with asymmetrical task set is the submarine combat system [24] (Fig. 2).

7.2. Performance evaluation parameters

We calculate the percentage of excess energy consumption (δE) of the EETSA algorithm as compared to the OPT algorithm as follows:

$$\delta E = 100(E_{EETSA}/E_{OPT} - 1), \tag{17}$$

where E_{EETSA} is the energy consumption of the EETSA algorithm, and E_{OPT} is the energy consumption of the OPT algorithm. For performance evaluation we take the average value of δE for a given set of task graphs. We also compare the running times of the algorithms. Besides these we also use some statistical parameters like average ($\bar{X}$), and standard deviation (σ_X) [32]. We calculate the percentage of optimal solutions (*pos*) that is the number of times the EETSA algorithm will give optimal solutions if it is run 100 times.

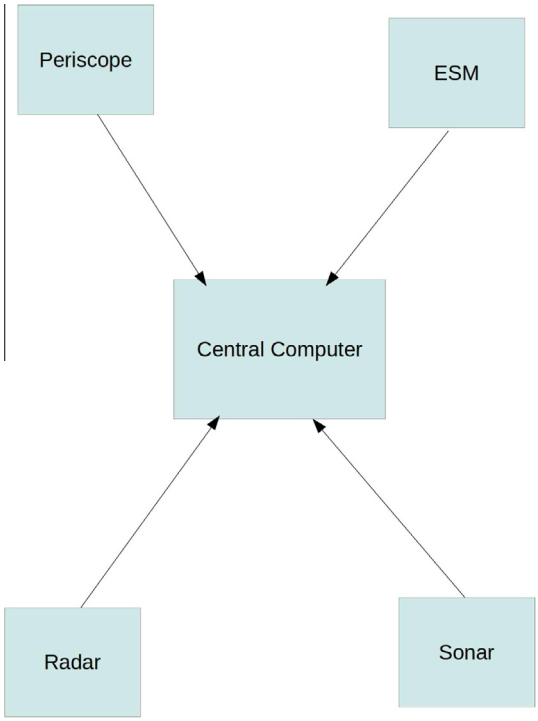

Fig. 2. Submarine combat system: a system with asymmetrical task set.

Table 1
Energy consumption and running time (in seconds) for symmetrical task sets for $p = q = 2$.

#	t	c	D	$\bar{E}_{EETSA}$	$\sigma_{E_{EETSA}}$	E_{OPT}	$\bar{\tau}_{EETSA}$	$\sigma_{\tau_{EETSA}}$	τ_{OPT}	*pos*
1	5	15	30	250.5	5.12	213	0.000144	0	0.54	0
2	5	20	30	400.0	0.00	400	0.000007	0	0.51	100
3	6	20	30	480.0	0.00	480	0.000007	0	1.23	100
4	6	25	40	570.0	0.00	570	0.000124	0	4.52	100
5	7	25	50	704.0	0.00	704	0.000011	0	29.51	100
6	7	30	60	840.0	0.00	840	0.000014	0	70.40	100
7	8	10	30	270.8	5.50	200	0.000151	0	6.35	0
8	8	15	30	480.0	0.00	480	0.000007	0	6.09	100
9	9	15	40	513.0	0.00	513	0.000123	0	50.47	100
10	9	20	50	720.0	0.00	720	0.000012	0	145.55	100
11	10	15	40	570.0	0.00	570	0.000123	0	110.66	100
12	10	20	50	800.0	0.00	800	0.000011	0	317.57	100

7.3. Results for symmetrical task sets

Table 1 shows the results for symmetrical task sets. Column '#' gives the serial number of the task set. Column 't' gives the number of tasks in the task set. Column 'c' gives the computational requirement of each task in the task set. Column 'D' gives the deadline of the task set. Column '$\bar{E}_{EETSA}$' gives the average energy consumption of the EETSA algorithm. Column '$\sigma_{E_{EETSA}}$' gives the standard deviation of energy consumption of the EETSA algorithm. Column 'E_{OPT}' gives the energy consumption of the OPT algorithm. Column '$\bar{\tau}_{EETSA}$' gives the average running time (in seconds) of the EETSA algorithm. Column '$\sigma_{\tau_{EETSA}}$' gives the standard deviation of running time of the EETSA algorithm. Column 'τ_{OPT}' gives the running time (in seconds) of the OPT algorithm. Column '*pos*' gives the percentage of optimal solutions for the EETSA algorithm.

For symmetrical task sets we have taken 12 task sets with $5 \leqslant t \leqslant 10, 10 \leqslant c \leqslant 30$, and $30 \leqslant D \leqslant 60$. From the table we can see that the EETSA algorithm is running very fast as compared to the OPT algorithm, and also that it is giving optimal solutions except for 2 task sets.

7.4. Results for asymmetrical task sets

We take the deadline of the asymmetrical task sets as $D = 30$. We take a total of 180 randomly generated task sets with the number of tasks t as 5, 6, 7, 8, 9, and 10 respectively, each having 30 task sets. For the task sets with the number of tasks as 5, 6, and 7, we randomly generate tasks having computational requirement between 10 and 20. This is representative of task sets having low variation in computational requirements of tasks (ratio between maximum to minimum computational requirement is only $2:1$). For the task sets with the number of tasks as 8, 9, and 10, we randomly generate tasks having computational requirements between 1 and 20. This is representative of task sets having high variation in computational requirements of tasks (ratio between maximum to minimum computational requirement is $20:1$).

Figs. 3–8 show a comparison of average percentage of excess energy consumption between the EETSA algorithm and the OPT algorithm for task sets having 5, 6, 7, 8, 9, and 10 tasks respectively. Figs. 3–5 compare the two algorithms for tasks having computational requirements between 10 and 20. The average percentage of excess energy consumption of EETSA over OPT are 56.14% for 5-node task sets, 16.01% for 6-node task sets, and 6.67% for 7-node task sets. We observe that as we increase the number of tasks, the EETSA algorithm gives results closer to the OPT algorithm. This is as expected because when we increase the number of tasks, we are actually decreasing the number of schedules that are within the deadline. Figs. 6–8 compare the two algorithms for tasks having computational requirements between 1 and 20. The average percentage of excess energy consumption of EETSA over OPT are 50.91% for 8-node task sets, 24.51% for 9-node task sets, and 10.31% for 10-node task sets. As observed before, as we increase the number of tasks, the EETSA algorithm gives results closer to the OPT algorithm. Also the performance of the EETSA algorithm decreases for the task sets having 8, 9, and 10 tasks as compared to the task sets having 5, 6, and 7 tasks. This is as expected because we have to increase the values of m and n for larger task sets to get comparable results. Average *pos* of EETSA is 0.63% for 5-node task sets, 0.67% for 6-node task sets, 1.83% for 7-node task sets, 5.27% for 8-node task sets, 11.23% for 9-node task sets, and 26.40% for 10-node task sets.

Figs. 9–14 show a comparison of running time (in seconds) between the EETSA algorithm and the OPT algorithm for task sets having 5, 6, 7, 8, 9, and 10 tasks respectively. Figs. 9–11 compare the two algorithms for tasks having computational requirements between 10 and 20. In Fig. 9, average running time of EETSA ranges from 0.000133 s to 0.000200 s with an average of 0.000161 s. Running time of OPT ranges from 0.507283 s to 0.773465 s with an average of 0.561891 s. In Fig. 10, average running time of EETSA ranges from 0.000025 s to 0.000159 s with an average of 0.000137 s. Running time

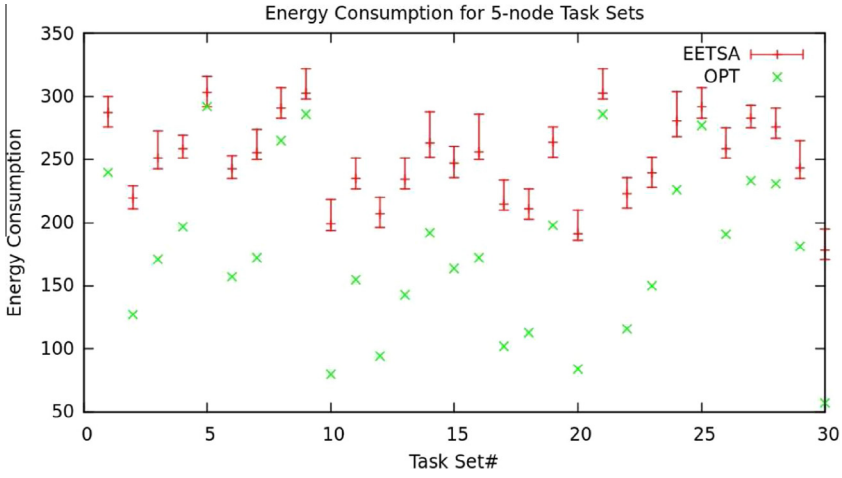

Fig. 3. Energy consumption for 5-node task sets. EETSA is consuming 56.14% more energy than OPT. Average *pos* of EETSA is 0.63%.

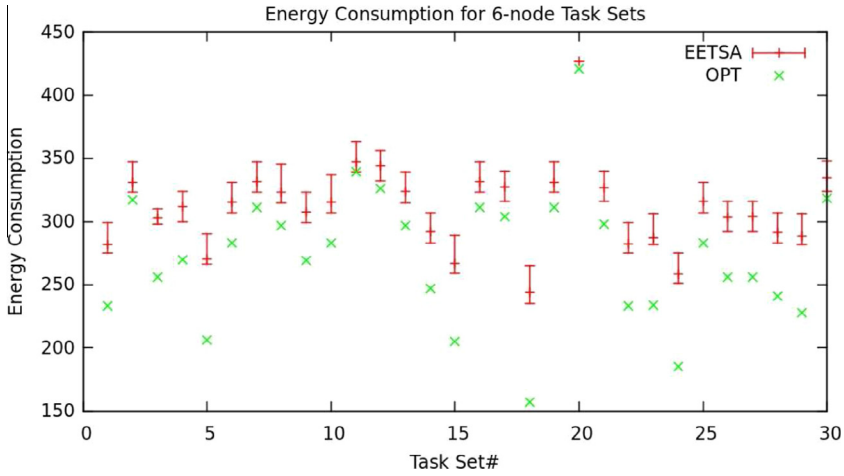

Fig. 4. Energy consumption for 6-node task sets. EETSA is consuming 16.01% more energy than OPT. Average *pos* of EETSA is 0.67%.

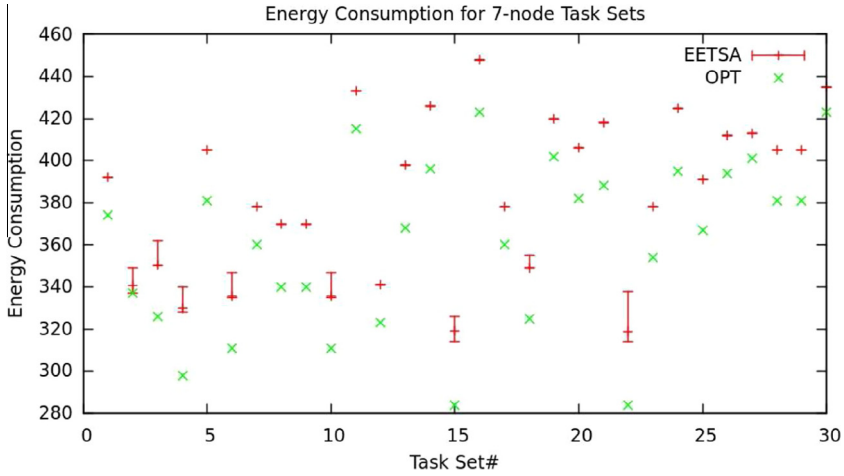

Fig. 5. Energy consumption for 7-node task sets. EETSA is consuming 6.67% more energy than OPT. Average *pos* of EETSA is 1.83%.

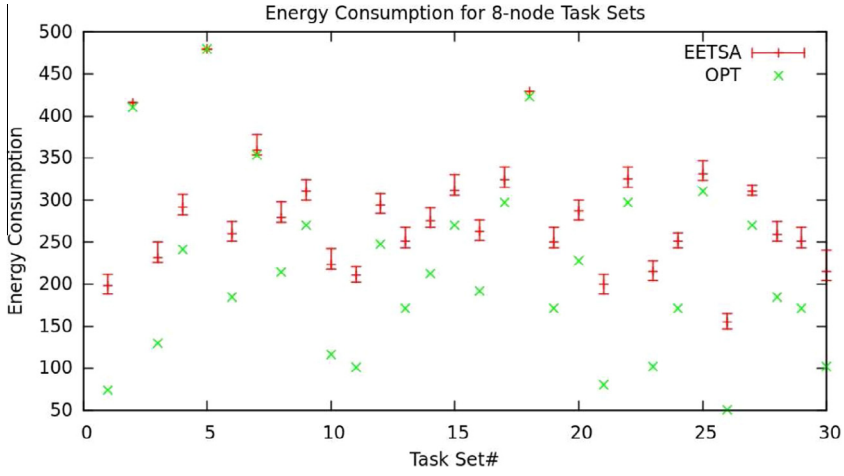

Fig. 6. Energy consumption for 8-node task sets. EETSA is consuming 50.91% more energy than OPT. Average *pos* of EETSA is 5.27%.

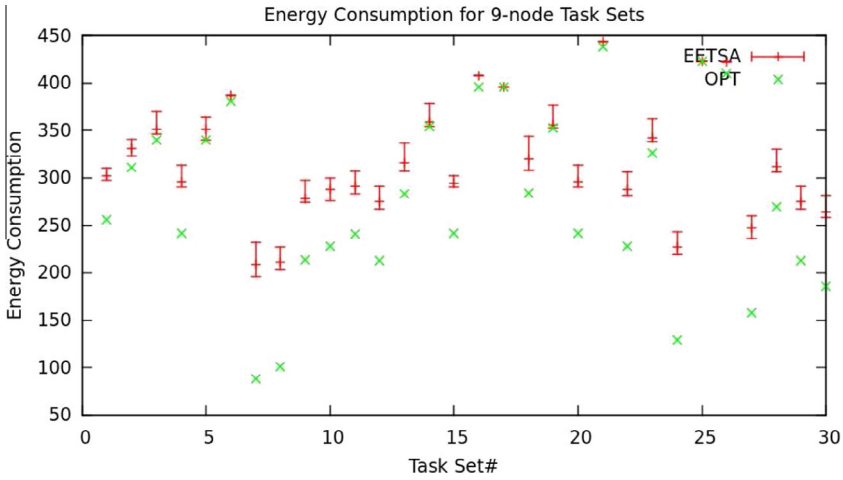

Fig. 7. Energy consumption for 9-node task sets. EETSA is consuming 24.51% more energy than OPT. Average *pos* of EETSA is 11.23%.

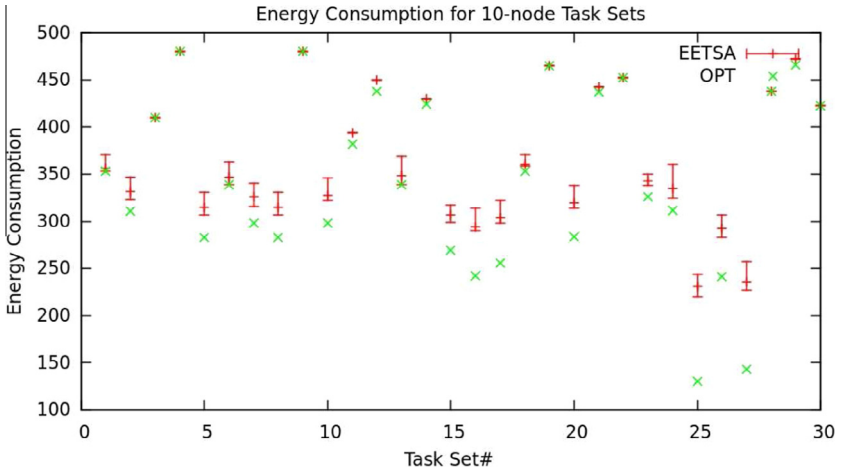

Fig. 8. Energy consumption for 10-node task sets. EETSA is consuming 10.31% more energy than OPT. Average *pos* of EETSA is 26.40%.

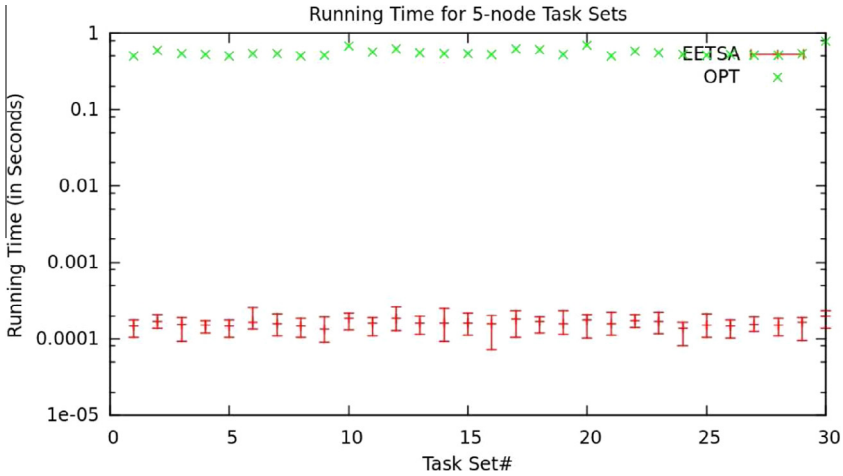

Fig. 9. Running time for 5-node task sets.

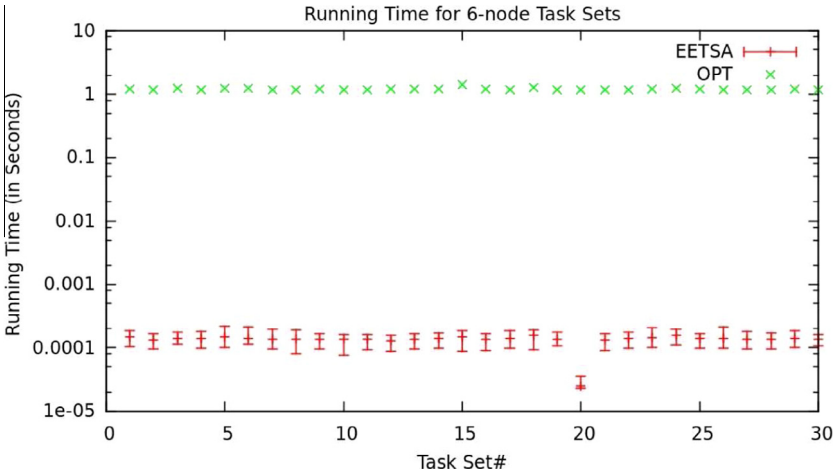

Fig. 10. Running time for 6-node task sets.

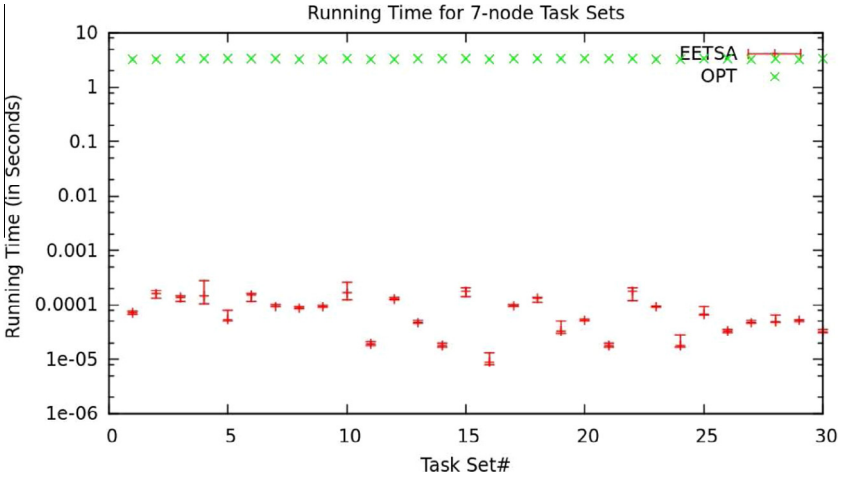

Fig. 11. Running time for 7-node task sets.

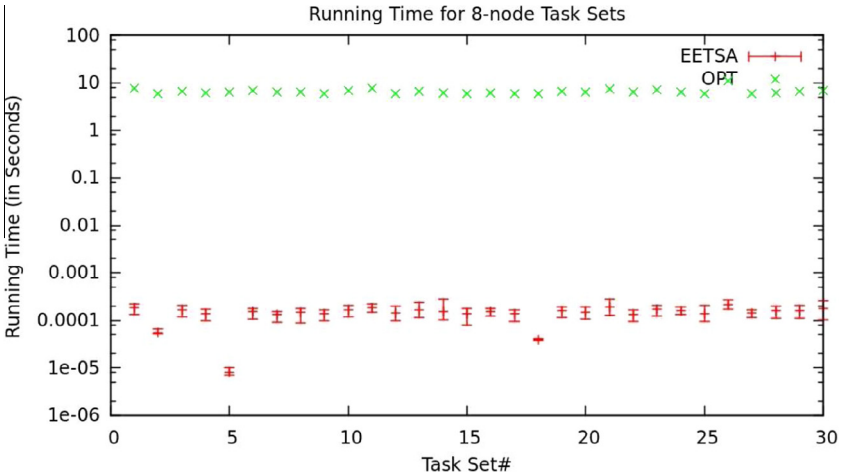

Fig. 12. Running time for 8-node task sets.

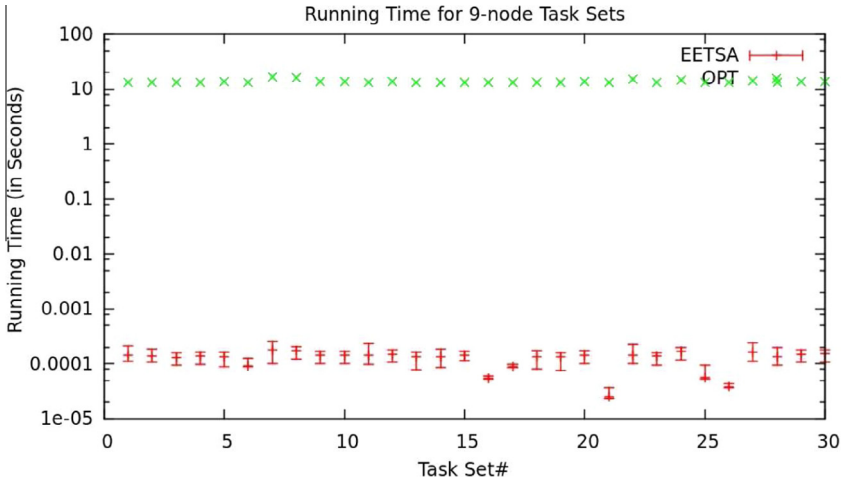

Fig. 13. Running time for 9-node task sets.

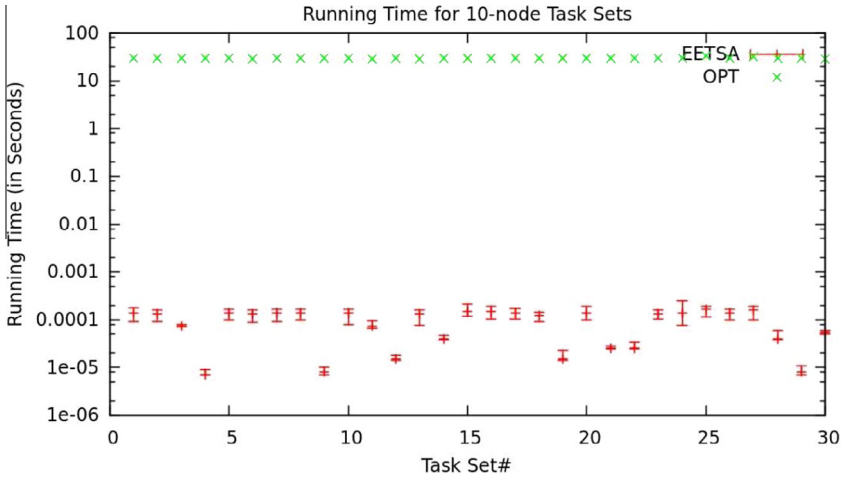

Fig. 14. Running time for 10-node task sets.

of OPT ranges from 1.157719 s to 1.432819 s with an average of 1.200884 s. In Fig. 11, average running time of EETSA ranges from 0.000009 s to 0.000180 s with an average of 0.000082 s. Running time of OPT ranges from 3.309868 s to 3.406952 s with an average of 3.347249 s. Figs. 12–14 compare the two algorithms for tasks having computational requirements between 1 and 20. In Fig. 12, average running time of EETSA ranges from 0.000008 s to 0.000213 s with an average of 0.000146 s. Running time of OPT ranges from 5.946618 s to 11.179679 s with an average of 6.654900 s. In Fig. 13, average running time of EETSA ranges from 0.000025 s to 0.000178 s with an average of 0.000128 s. Running time of OPT ranges from 13.121674 s to 16.540721 s with an average of 13.700316 s. In Fig. 14, average running time of EETSA ranges from 0.000007 s to 0.000167 s with an average of 0.000097 s. Running time of OPT ranges from 29.259462 s to 33.374651 s with an average of 30.053799 s.

8. Experimental results for quad-core processors

In this section we compare the EETSA algorithm with the OPT algorithm for quad-core processors. For the quad-core processor we take $\alpha = 1$, and $p = 4$, and $q = 2$. For the EETSA algorithm we take $m = n = 10$. The EETSA algorithm is run 100 times, and in the results we give the average and the standard deviation of energy consumed and the running time.

8.1. Results for symmetrical task sets

Table 2 shows the results for symmetrical task sets. For symmetrical task sets we have taken 10 task sets with $3 \leqslant t \leqslant 7, 4 \leqslant c \leqslant 8$, and $3 \leqslant D \leqslant 8$. From the table we can see that the EETSA algorithm is running very fast as compared to the OPT algorithm, and also that it is giving optimal solutions except for 4 task sets.

8.2. Results for asymmetrical task sets

We take the deadline of the asymmetrical task sets as $5 \leqslant D \leqslant 7$. We take a total of 150 randomly generated task sets with the number of tasks t as 3, 4, 5, 6, and 7 respectively, each having 30 task sets. We randomly generate tasks having computational requirement between 1 and 10. For the task sets with the number of tasks as 3, 4, and 5, we take the deadline as 5. For the task sets with the number of tasks as 6, we take the deadline as 6, and for the task sets with the number of tasks as 7, we take the deadline as 7.

Figs. 15–19 show a comparison of average percentage of excess energy consumption between the EETSA algorithm and the OPT algorithm for task sets having 3, 4, 5, 6, and 7 tasks respectively. The average percentage of excess energy consumption of EETSA over OPT are 5.95% for 3-node task sets, 4.27% for 4-node task sets, and 1.98% for 5-node task sets. We observe that as we increase the number of tasks for the same deadline, the EETSA algorithm gives results closer to the OPT algorithm. This is as expected because when we increase the number of tasks for the same deadline, we are actually decreasing the number of schedules that are within the deadline. The average percentage of excess energy consumption of EETSA over OPT are 4.08% for 6-node task sets, and 6.14% for 7-node task sets. The increase in average percentage of excess energy consumption of EETSA over OPT is due to increase in the deadline, because when we increase the deadline, we are actually increasing the number of schedules that are within the deadline. Average *pos* of EETSA is 80.23% for 3-node task sets, 79.27% for 4-node task sets, 79.60% for 5-node task sets, 57.37% for 6-node task sets, and 58.63% for 7-node task sets.

Figs. 20–24 show a comparison of running time (in seconds) between the EETSA algorithm and the OPT algorithm for task sets having 3, 4, 5, 6, and 7 tasks respectively. In Fig. 20, average running time of EETSA ranges from 0.000007 s to 0.000082 s with an average of 0.000041 s. Running time of OPT ranges from 0.050763 s to 0.117588 s with an average of 0.069965 s. In Fig. 21, average running time of EETSA ranges from 0.000007 s to 0.000078 s with an average of 0.000040 s. Running time of OPT ranges from 0.258776 s to 0.432593 s with an average of 0.282603 s. In Fig. 22, average running time of EETSA ranges from 0.000007 s to 0.000084 s with an average of 0.000031 s. Running time of OPT ranges from 1.247445 s to 1.485754 s with an average of 1.282587 s. In Fig. 23, average running time of EETSA ranges from 0.000012 s to 0.000226 s with an average of 0.000136 s. Running time of OPT ranges from 18.101837 s to 31.271310 s with an average of 18.874743 s. In Fig. 24,

Table 2
Energy consumption and running time (in seconds) for symmetrical task sets for $p = 4$, and $q = 2$.

#	t	c	D	$\bar{E}_{EETSA}$	$\sigma_{E_{EETSA}}$	E_{OPT}	$\bar{\tau}_{EETSA}$	$\sigma_{\tau_{EETSA}}$	τ_{OPT}	pos
1	3	6	3	72.00	0.00	72	0.000004	0	0.013245	100
2	3	6	6	29.52	9.01	18	0.000090	0	0.222119	37
3	4	4	8	16.72	4.09	16	0.001747	0.001	9.76	97
4	4	5	8	20.96	4.70	20	0.001154	0	7.16	96
5	5	5	8	39.94	5.79	37	0.000751	0	28.93	67
6	5	6	6	120.00	0.00	120	0.000016	0	4.82	100
7	6	6	6	144.00	0.00	144	0.000013	0	22.58	100
8	6	7	8	150.00	0.00	150	0.000494	0	122.55	100
9	7	7	7	200.00	0.00	200	0.000019	0	284.68	100
10	7	8	8	224.00	0.00	224	0.000021	0	563.74	100

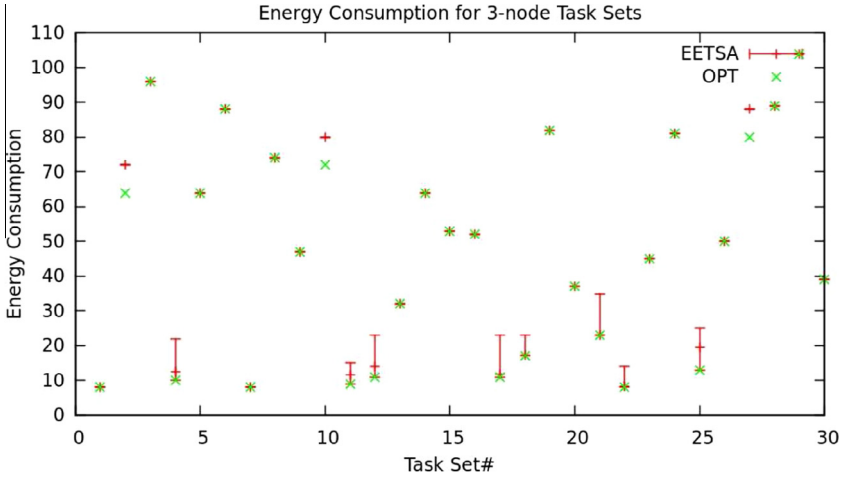

Fig. 15. Energy consumption for 3-node task sets. EETSA is consuming 5.95% more energy than OPT. Average *pos* of EETSA is 80.23%.

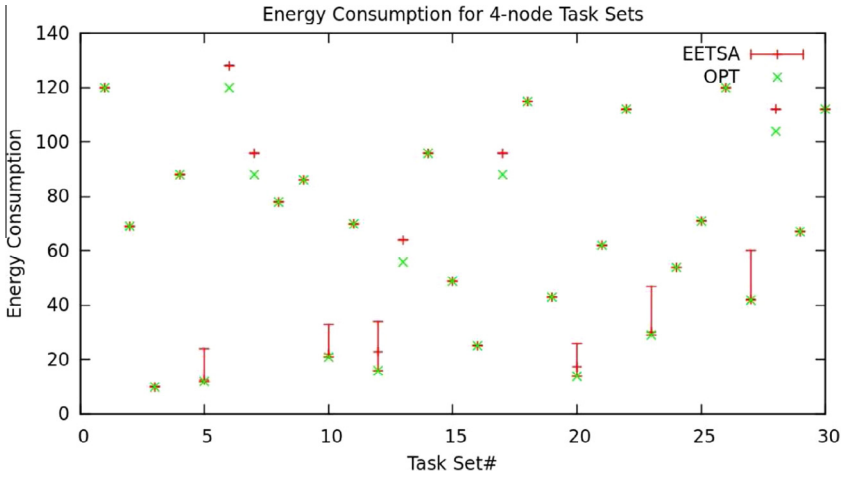

Fig. 16. Energy consumption for 4-node task sets. EETSA is consuming 4.27% more energy than OPT. Average *pos* of EETSA is 79.27%.

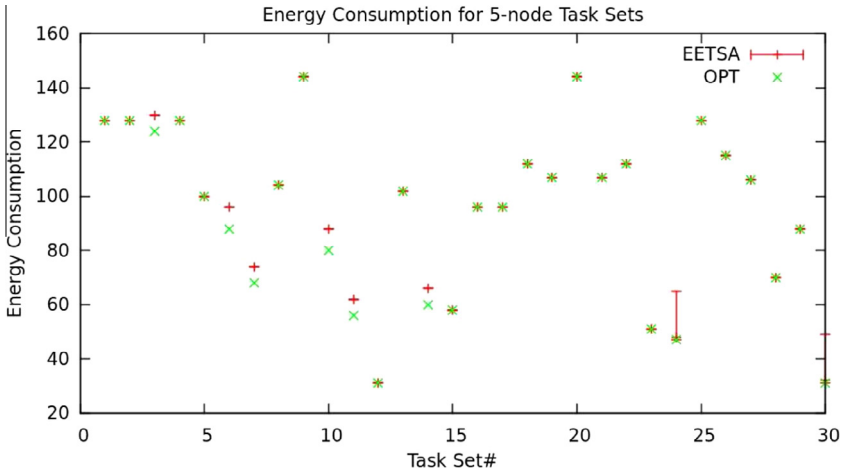

Fig. 17. Energy consumption for 5-node task sets. EETSA is consuming 1.98% more energy than OPT. Average *pos* of EETSA is 79.60%.

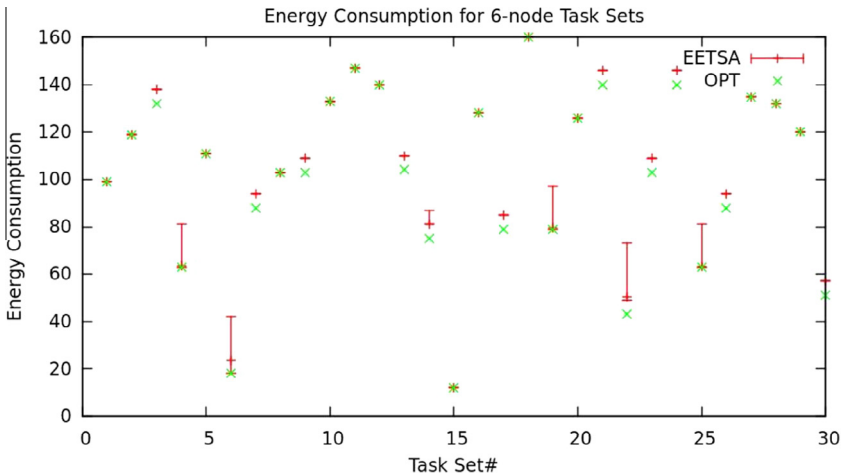

Fig. 18. Energy consumption for 6-node task sets. EETSA is consuming 4.08% more energy than OPT. Average *pos* of EETSA is 57.37%.

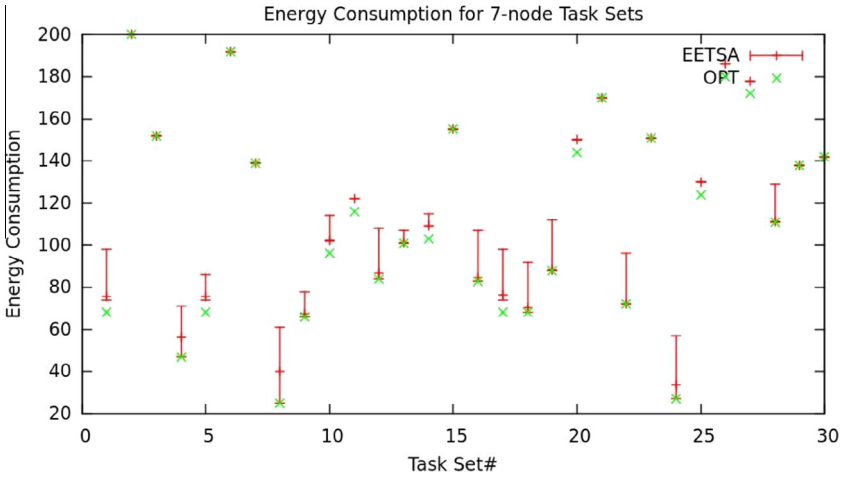

Fig. 19. Energy consumption for 7-node task sets. EETSA is consuming 6.14% more energy than OPT. Average *pos* of EETSA is 58.63%.

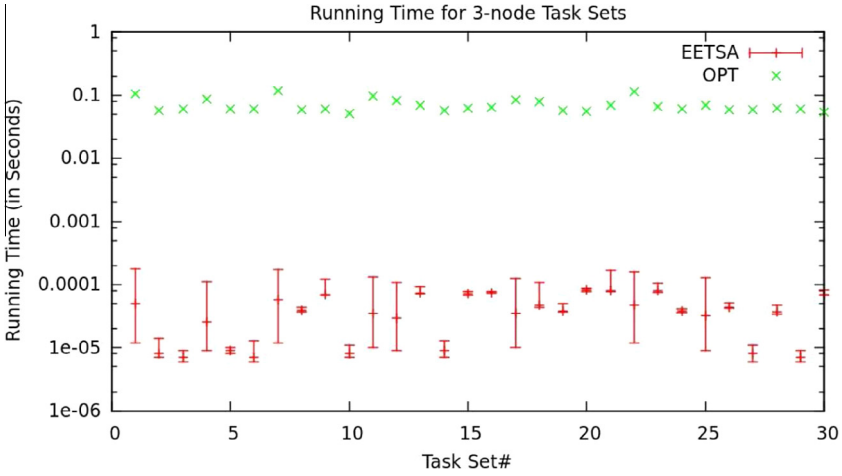

Fig. 20. Running time for 3-node task sets.

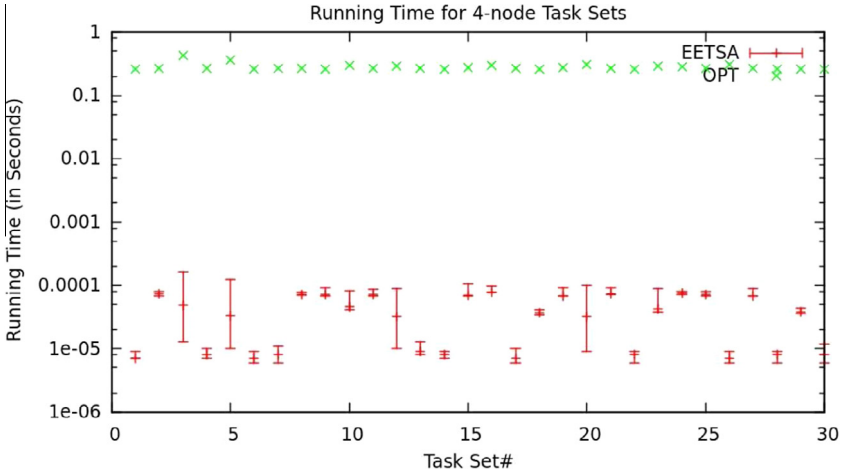

Fig. 21. Running time for 4-node task sets.

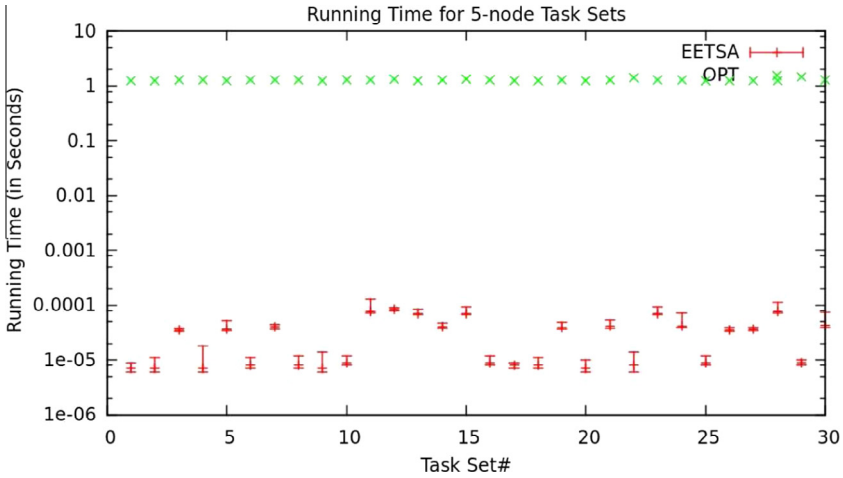

Fig. 22. Running time for 5-node task sets.

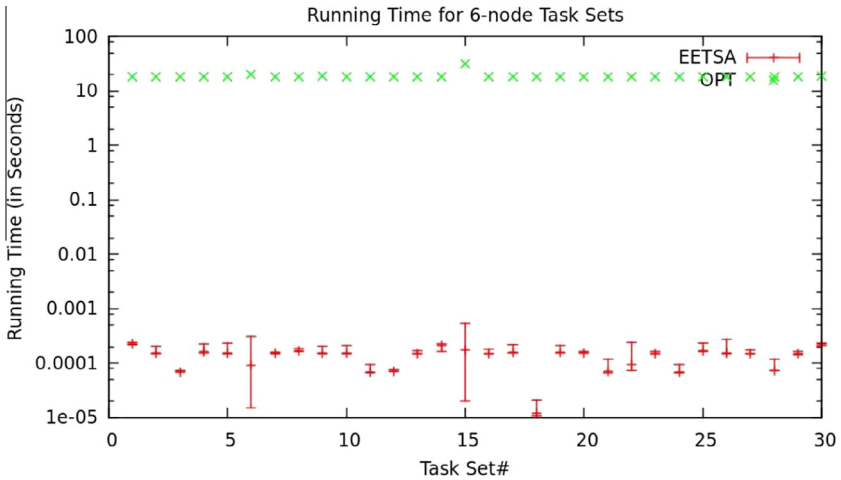

Fig. 23. Running time for 6-node task sets.

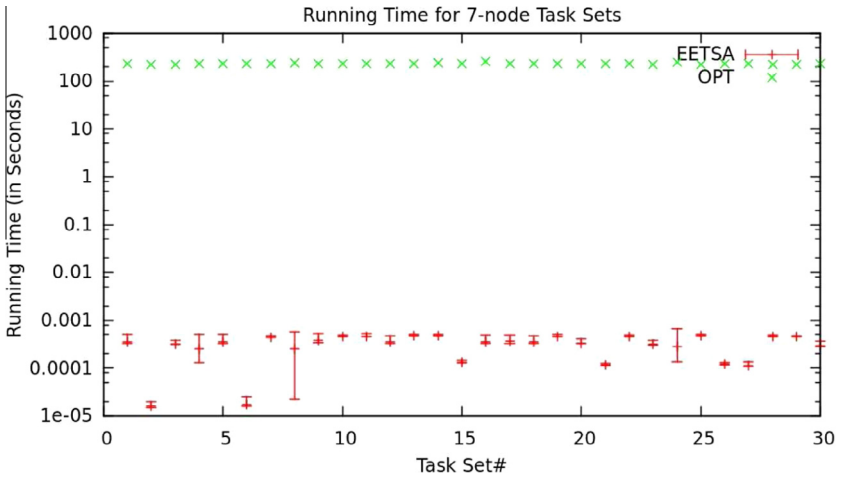

Fig. 24. Running time for 7-node task sets.

average running time of EETSA ranges from 0.000016 s to 0.000473 s with an average of 0.000323 s. Running time of OPT ranges from 227.437292 s to 259.608653 s with an average of 232.199073 s.

9. Conclusions

We considered a system with a single multi-core processor with software controlled DVS having a finite set of core speeds and discussed a task scheduling problem associated with it. The problem that we pondered was to find a minimum energy task scheduling for a given set of independent tasks that have to be completed within a given common deadline. We proposed a Monte Carlo algorithm (the EETSA algorithm) of complexity $O(t(mp+q+log(t))+p(t+q)(D^{pq}+n))$ (here t is the number of tasks, p is the number of cores, q is the number of core speeds, m is an integer parameter that is the number of iterations we should try to get a feasible solution before declaring that no solution is possible, n is an integer parameter that is the number of iterations we should try to reduce the energy consumption when we get a feasible solution, and D is the common deadline of the tasks) for solving the task scheduling problem and compared it with the optimal algorithm (the OPT algorithm) for both symmetrical and asymmetrical task sets. Our finding was that the EETSA algorithm is very fast as compared to the OPT algorithm, it gives almost optimal results for symmetrical task sets, and that it gives comparable results for the case of asymmetrical task sets. For future work we should try some more heuristics for solving the EETSP problem.

Acknowledgements

The authors are thankful to the anonymous referees for valuable comments and suggestions in revising the manuscript to the present form.

References

[1] J. Fruehe, Planning considerations for multi-core processor technology, Dell Power Solutions, May 2005, pp. 67–72.
[2] B. Schauer, Multi-core processors – a necessity, ProQuest Discovery Guides, September 2008, pp. 1–14.
[3] P. Pillai, K.G. Shin, Real-time dynamic voltage scaling for low-power embedded operating systems, in: ACM Symp. Operating Syst. Principles, 2001, pp. 89–102.
[4] C.Y. Yang, J.J. Chen, T.W. Kuo, An approximation algorithm for energy-efficient scheduling on a chip multiprocessor, in: Proc. Design Autom. Test in Eur. Conf. Exhibition, 2005, pp. 468–473.
[5] H. Aydin, R. Melhem, D. Mosse, P. Mejia-Alvarez, Dynamic and aggressive scheduling techniques for power-aware real-time systems, in: Proc. 22'nd IEEE Real-Time Syst. Symp., 2001, pp. 95–105.
[6] J.J. Chen, T.W. Kuo, C.L. Yang, Profit-driven uniprocessor scheduling with energy and timing constraints, in: ACM Symp. Appl. Comput., 2004, pp. 834–840.
[7] J.J. Chen, T.W. Kuo, H.I. Lu, Power-saving scheduling for weakly dynamic voltage scaling devices, Algorithms and Data Struct. Lecture Notes in Comput. Sci. 3608 (2005) 338–349.
[8] T. Ishihara, H. Yasuura, Voltage scheduling problems for dynamically variable voltage processors, in: Proc. 1998 Int. Symp. Low Power Electron. Des., 1998, pp. 197–202.
[9] P. Mejia-Alvarez, E. Levner, D. Mosse, Adaptive scheduling server for power-aware real-time tasks, ACM Trans. Embedded Comput. Syst. 3 (2004) 284–306.
[10] F. Yao, A. Demers, S. Shenker, A scheduling model for reduced CPU energy, in: Proceedings of the 36'th Annual Symposium on Foundations of Computer Science, IEEE, 1995, pp. 374–382.
[11] H.S. Yun, J. Kim, On energy-optimal voltage scheduling for fixed-priority hard real-time systems, ACM Trans. Embedded Comput. Syst. 2 (2003) 393–430.
[12] J.H. Anderson, S.K. Baruah, Energy-efficient synthesis of periodic task systems upon identical multiprocessor platforms, in: Proc. 24'th Int. Conf. Distrib. Comput. Syst., 2004, pp. 428–435.
[13] J.J. Chen, H.R. Hsu, K.H. Chuang, C.L. Yang, A.C. Pang, T.W. Kuo, Multiprocessor energy-efficient scheduling with task migration considerations, in: Proc.16'th Euromicro Conf. Real-Time Syst., 2004, pp. 101–108.
[14] F. Gruian, System-level design methods for low-energy architectures containing variable voltage processors, Power-Aware Comput. Syst. (2000) 1–12.
[15] F. Gruian, K. Kuchcinski, Lenes: task scheduling for low energy systems using variable supply voltage processors, in: Proc. Asia South Pacific Design Automation Conf., 2001, pp. 449–455.
[16] R. Mishra, N. Rastogi, D. Zhu, D. Mosse, R. Melhem, Energy aware scheduling for distributed real-time systems, in: Int. Parallel Distrib. Process. Symp., 2003, pp. 21.
[17] Y. Zhang, X. Hu, D.Z. Chen, Task scheduling and voltage selection for energy minimization, in: Annual ACM IEEE Design Autom. Conf., 2002, pp. 183–188.
[18] D. Zhu, R. Melhem, B. Childers, Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor realtime systems, in: Proc. IEEE 22'nd Real-Time Syst. Symp., 2001, pp. 84–94.
[19] S. Irani, S. Shukla, R. Gupta, Algorithms for power savings, in: Proc. 14'th Annual ACM-SIAM Symp. Discrete Algorithms Soc. Ind. Appl. Math., 2003, pp. 37–46.
[20] Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor, White Paper, Intel Corporation, March 2004.
[21] AMD PowerNow! technology, informational white paper, Advanced Micro Devices Inc., 2000.
[22] J.J. Chen, C.F. Kuo, Energy efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms, in: 13'th IEEE Int. Conf. Embedded and Real-Time Comput. Syst. Appl., 2007, pp. 28–38.
[23] Y. Liu, H. Liang, K. Wu, Scheduling for energy efficiency and fault tolerance in hard real-time systems, in: Proc. Des. Autom. Test in Eur. Conf. Exhibition, 2010, pp. 1444–1449.
[24] Raytheon, AN/BYG-1(V) Submarine Combat System, <www.raytheon.com/businesses/rtnwcm/groups/public/documents/content/rtnbusidsprodanbyg1pdf.pdf>, 2007.
[25] W. Lin, Video Surveillance, InTech, Croatia, 2011.
[26] A. Fukuda, Tracking migratory birds using GPS, Nature Interface 6 (2001) (2001) 18–19.
[27] A. Mishra, A.K. Tripathi, On the complexity of scheduling independent tasks with common deadline on multi-core processors with software controlled dynamic voltage scaling, unpublished manuscript, 2013.
[28] E. Horowitz, S. Sahni, S. Rajasekaran, Fundamentals of Computer Algorithms, W.H. Freeman, 1998.

[29] E.W. Briao, D. Barcelos, F.R. Wagner, Dynamic task allocation strategies in MPSoC for soft real-time applications, in: Proc. Design Autom. Test in Europe Conf. Exhibition, 2008, pp. 1386–1389.
[30] D.S. Hochbaum, Approximation Algorithms for NP-Hard Problems, PWS Publishing Company, 1997.
[31] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, The MIT Press, 2001.
[32] P.K. Mishra, A. Mishra, K.S. Mishra, A.K. Tripathi, Benchmarking the clustering algorithms for multiprocessor environments using dynamic priority of modules, Appl. Math. Model. 36 (2012) 6243–6263.