Multiprocessor Procrastination Scheduling in ERFair Realtime Systems

Anshuman Tripathi[1]

Dept. of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Abstract

Processor shutdown is primary method to reduce the leakage power in ideal processors. This paper presents, ERFair Processor Procastination Scheduler (EPPS), an ERFair scheduler for multiprocessor, hard-realtime systems that attempts to locally maximize the average processor shutdown interval while maintaining proportional fairness for the tasks. Comparative studies with the Basic-ERFair scheduler and ESPS algorithm show that EPPS algorithm can achieve as high as **10** times the shutdown length achievable by these uniprocessor counter parts.

Keywords: ERFair Scheduler, Procrastination, Processor shutdown, multiprocessor.

1. Introduction

The presence of embedded systems like PDAs, cellphones, etc, powered by sources of limited energy, has proliferated in contemporary technology. This combined with the leakage power loss in gate chips has made the problem of power dissipation and energy efficiency critical to design of devices and software.

While there are many advancements in reducing the power consumption in at the level of processor design, with the advent of multiple power states in processors many software based optimizations have become possible. From [1] the power consumed in a processors can be crudely described by the following equation:

$$\underbrace{C \times s \times V^2}_{\text{dynamic power}} + \overbrace{\underbrace{V \times I_{sc}}_{\text{short circuit power}} + \underbrace{V \times I_{leak}}_{\text{leakage power}}}^{\text{static power}} \tag{1}$$

where s is the frequency of operation for the CPU. Based on equation 1, there are two major approaches that are commonly used for power reduction viz: Dynamic voltage scaling (DVS) [2, 3, 4] and processor shutdown [5, 6]. In DVS technology, a processor has a set of operating frequencies mapped to the supply voltage. As evident from Equation 1, a reduction in s and V leads to lower power dissipation. Processor shutdown on the other hand means putting the processor in a sleep/OFF power state, where the processor can not be used for any further computations. In this suspended state, the processor consumes very less energy and can be again put to normal state, subject to loss of power and time in the transition. In both these approaches the transition of processors among states consumes power. Thus a practical use of this technology will require maximizing the average length of time CPU is in a particular state (to make the transition to that state profitable).

This paper proposes an approach for efficient processor shutdown in ERFair [7] multiprocessor systems. ERFair scheduling is an important approach in QoS applications, as it concurrently handles many independent aspects [8, 9, 10] like realtime audio-video, telnet, gaming, web browsing, etc. This paper presents *ERFair Processor Procrastination Scheduler (EPPS)* a modified version of the basic ERFair scheduler, that greedily shuts down processors while locally maximizing the shutdown intervals.

The rest of the paper is organized as following: In Section 2 a brief comparison of previous works has been presented, Section 3 describes the system model that is considered in the design of the algorithm, it gives a brief description of the power model used and the ERFair systems. A detailed analysis and description of *EPPS* algorithm is given in Section 4. The implementation and simulation results for different datasets is presented in Section 5. Finally Section 6, gives a summary of the conclusions.

2. Related Work

As described in previous section, practical use of power down technology for energy savings, requires idle period to be long enough so as to amortize the transition costs. A completely greedy approach, therefore, is not necessarily fruitful as it may degrade the performance without appreciable improvements to energy consumption. Previous works like [11, 12] attempt to maximize the idle time interval by delaying the execution of tasks. Author in [13] proposes a polynomial time algorithm ($O\left(n^5\right)$) to find the time intervals, when the processors could be shutdown, using a dynamic programming approach. There are also works where a combination of DVS and power down technologies has been considered, like [14, 15, 16, 17, 18, 19]. PACE algorithm described in [20] proposes a generic approach of varying the operating frequency in a DVS enabled processors, to maximize the power gains. It may be noted that use

Email address: anshumant.iit@gmail.com (Anshuman Tripathi)

of DVS in combination with shutdown technology in general gives a better energy saving as compared to using the DVS technology alone, since slowdown causes increase in the leakage power.

ESPS algorithm described in [5] proposes a novel approach of slack stealing in ERfair systems by modifying the basic ERFair algorithm and scheduling processor shutdowns to locally maximize the intervals in uniprocessor systems. The present work develops on similar lines by extending the scope of shutdown to multiprocessor environments. In the present work we compare the performance of both ESPS and the proposed algorithm in multiprocessor systems.

3. System Model

It is assumed that the system has multiple identical processors and the tasks submitted to the system are periodic tasks with hard deadlines. The scheduler attempts to put processors in power saving state, while still being ERFair to the processes. In this section a brief introduction of ERFair scheduling is provided, along with the Energy model for the processors that dictates the design of scheduler.

3.1. ERFair Scheduling

ERFair scheduling falls in the category of proportionally fair scheduling approach, which is an effective strategy for rate-based realtime applications. A rate based application, other than the deadline, also requires minimum guaranteed quality of service of the form *reserve X units of time for application A out of every Y units*.

Consider a set of tasks $\{T_1, T_2, \dots T_n\}$, such that the computational requirement of T_i is e_i and the period for recurrence of task instance is p_i. For such a task set, an ERFair Scheduler not only aims to meet the deadlines of the tasks, but also ensures a minimum rate of execution for the task. Such scheduler tries to schedule tasks such that in t time slots from arrival of task T_i, at least $\lfloor \frac{e_i \times t}{p_i} \rfloor$ units of execution time is given. To state more formally, *lag* of a task T_i is defined as:

$$lag(T_i) = \frac{e_i \times t}{p_i} - allocated(T_i, t) \tag{2}$$

where $allocated(T_i, t)$ is the actual execution time allotted to task T_i till time slot t. Based on this definition of *lag*, a scheduler is said to be ERFair iff:

$$\forall (t, i)\, lag(T_i, t) < 1 \tag{3}$$

This means that in an ERFair scheduler, if there is an under allocation for a task, it must always be less than one time slot. The criteria for ERFair schedulability of a task set is:

$$\sum_{1}^{n} \frac{e_i}{p_i} \leq m \tag{4}$$

Equation 4 simply translates to the obvious condition that the total utilization of tasks in the task set, should not be any more than the number of processors available. The ratio $\frac{e_i}{p_i}$ is called the *weight* of task T_i and the summation $\sum_{1}^{n} \frac{e_i}{p_i}$ is called the utilization of the task set. Thus for ERFair schedulability of a task set, condition in equation 4, signifies that total task utilization be less than available processors.

3.2. Energy Model

Although shutting down processors can save on their power consumption, there are other factors that limit the minimum amount of time a processor should be shut down for it to be beneficial. The power state transitions consume extra power, so the time of shut down should be long enough to compensate for the power requires in state transitions. The minimum length of a profitable processors shut down is called the break even time point $T_{breakpoint}$.

The exact value of $T_{breakpoint}$ is processor specific and is dependent on the Power used in ON state (P_{ON}), the Power requirement for transitioning (P_{tr}), the Power used when processor is in sleep state (P_{OFF}) and the time taken for transition from ON to SLEEP state (Ttr). Based on these architecture specific variables, the break even time can be formulated as:

$$T_{breakpoint} = T_{tr} + T_{tr} \times \frac{P_{tr} - P_{ON}}{P_{ON} - P_{OFF}} \tag{5}$$

4. EPPS algorithm

4.1. Working Principle

As described in [5], the slack of a task is the amount of time it can be suspended from the active queue without any possibility of violating ERFairness of the scheduler. It is also noted that for tasks in ready queue the slack increases uniformly at the rate of $\frac{1-w}{w}$ where w is the total utilization of the system. Although these claims have been made for uniprocessor systems in [5], it can nevertheless be generalized for multiprocessor systems (as later proved in Theorem 1). The *ERFair Processor Procrastination Scheduler (EPPS)* works by using the accumulated system slack as a leverage to greedily shutdown processors, there by reducing their ideal time and subsequently the leakage power dissipation. It must be noted that when a new instance of a task arrives, the system slack always becomes zero, since the slack of new instance of a task is zero. Thus at any point in time the farthest time until which processors can be shutdown without the risk of violating ERFairness, is the closest expected arrival time (in case of our model its same as the closest deadline).

For a system with m processors and n tasks with total weight U, if at least $\lceil U \rceil$ processors are available in power on state, the schedulability criteria for ERFairness as defined in Equation 4 is satisfied. As a result $m - \lceil U \rceil$ processors can be shutdown without any risk of violating fairness to tasks. At any time t during the task schedule, if the number of processors active are m_a, then Algorithm 1 tries to speculate when the system slack will be high enough so that an extra processor can be shutdown (i.e only $m_a - 1$ processors will be active). Let us assume that the closest arrival time for system is t_d. Since the system slack at t_d is zero, all we require is that it remains positive before t_d. To maximize the shutdown interval, we should shutdown the extra processor at a time t_{cut} such that system slack gradually decreases and becomes zero at t_d. The trajectory taken by slack in this case is given by equation:

$$Slack(t) = r^{'} \times (t - t_d) \tag{6}$$

where $r^{'}$ is the rate of slack generation when system has $m_a - 1$ processors, which (according to [5]) is given by

$$r^{'} = \begin{cases} \frac{1-w^{'}}{w^{'}} & m_a > 1 \\ -1 & m_a = 1 \end{cases} \tag{7}$$

where $w^{'}$ is given as:

$$w^{'} = \frac{U}{m_a - 1} \tag{8}$$

By taking the intersection point of equation 6 with the slack trajectory of the task with minimum slack i.e (T_{min}) we can compute the time t_{cut}, when system can have only $m_a - 1$ processors. The shutdown would be practical only if $t_d - t_{cut} \geq T_{breakpoint}$.

4.2. Slack updates

The EPPS algorithm requires that the slack of all the tasks is updated regularly. This however is a very expensive operation, since one update of slacks will require at least $O(n)$ time, where n is the number of tasks in the system. To overcome this obstacle we use the following results. Since the system is ERFair, the slack rate r will be same amongst *Active* tasks and *Inactive* tasks. For any task T_i the slack rate r_i is given by:

$$r_i = \begin{cases} \frac{1-w}{w} & T_i \in Active \\ -1 & T_i \in Inactive \end{cases} \tag{9}$$

where w is the utilization of the system. Thus task slack only needs to be updated when the slack rate changes for the task, which is possible in three scenarios:

1. When a task completes its execution (i.e it is removed from *Active* and put to *Inactive* heap). This step not only changes the slack rate for finished task but since the utilization w is changed, slack rate of all the *Active* tasks is also affected.
2. When either a new instance of a task or a new task arrives to *Active* heap.
3. When a processor is shutdown or woken up (thus changing the value of w in equation 9).

Algorithm 1 thus only updates the slacks of the tasks in one of these three conditions.

4.3. Details

Based on the discussions in the previous sections, a detailed pseudo-code description of the algorithm for *EPPS* is given in Algorithm 1. The logic of the pseudo-code presented here is in line with the working principle of *EPPS* algorithm described in the previous section.

Algorithm 1 *Algorithm EPPS*

```
1: {Given: A set of $n$ tasks and $m$ processors.}
2: Power down processors so that only $\lceil U \rceil$ processors are ON.
3: for Each time slot $t$, do
4:     Select and execute the most urgent $m$ subtasks from the heap of Active tasks.
5:     Is a task has completed, then put it in Inactive tasks otherwise re-insert it in heap of Active tasks.
6:     Let sum of task weights (in Active heap) = $U$
7:     if A new task OR subtask has arrived then
8:         Update Task slacks
9:         Power up processors, so that $\lceil U \rceil$ processors are available in power-on state.
10:        $t_{shutdown}$ = NewShutdownPoint()
11:    if A task has finished then
12:        Update task slacks
13:        $t_{shutdown}$ = NewShutdownPoint()
14:    if $t == t_{shutdown}$ AND $m > 0$ then
15:        Update task slacks
16:        Shutdown 1 more processor
17:        $t_{shutdown}$ = NewShutdownPoint()
```

Algorithm 2 *Function NewShutdownPoint()*

```
1: {Given: A set of $n$ tasks and $m$ processors.}
2: let utilization of tasks in Active heap be $U$.
3: let there are $m_a$ processors currently active.
4: if $m_a == 1$ then
5:     $r = -1$
6: else
7:     $w = U/(m_a - 1)$ AND $r = \frac{1-w}{w}$
8: $t_d$ = Earliest deadline of for tasks in Active heap.
9: $T_{min}$ = Minimum Slack Task (amongst both Active and Inactive heap)
10: Find $t_{cut}$ the timeslot of intersection between Slack trajectory of $T_{min}$ and line $S = r \times (t - t_d)$. Where $S$ is slack variable and $t$ is timeslot variable
11: if $t_d$ - $t_{cut} \geq T_{breakpoint}$ then
12:     return return $t_{cut}$
13: else
14:     return return $-1$
```

4.4. Analysis

In this section some proofs are presented leading to the complexity analysis of *EPPS* as described in Algorithm 1.

Theorem 1. *The rate of slack generation for a task T_i in a multiprocessor environment is given by equation 9.*

PROOF. **Case** $T_i \in Inactive$: Since the task is in inactive state, it means its already suspended from execution. As per definition slack of the task T_i is the amount of time it can remain suspended without violating ERFairness. Since the task is already inactive, with every timeslot, the slack will decrease by 1 (since T_i has already spent 1 timeslot in suspended state). Thus for all inactive tasks the rate of slack generation is -1. The same argument holds good even when the number of processors in the system are 0.

Case $T_i \in Active$: The weight of task is given by $w_i = \frac{e_i}{p_i}$, let the sum of all the tasks be U. Since in Basic ERFair scheduling a processor is never left ideal, if there are m processors the rate of execution for the task is given by $w_i.\frac{m}{U}$. Thus the rate of overallocation is $w_i.\frac{m}{U} - w_i$. For each unit of overallocation, the overallocation in terms of time is $\frac{1}{w_i}$. Thus the rate of time overallocation is $\frac{m}{U} - 1$ which can be re-written as $\frac{1-w}{w}$.

Thus the slack generation is given by equation 9.

Theorem 2. *EPPS is ERFair.*

PROOF. It may be noted that EPPS as described in Algorithm 1 the slack of the system is never allowed to go below zero in event of processor shutdown.

Since the slack is the amount of time a task may be suspended from execution without violating ERFairness, there is no violations during the time of processor shutdown. In the remaining timeslots there are at least $\lceil U \rceil$ processors in the system, which satisfies the ERFair schedulability criteria.

Thus the system is ERFair in all timeslots.

Theorem 3. *The time complexity of a single call to* NextShutdownPoint() *is $O(n)$, where n is the number of tasks in the system.*

PROOF. In light of the description in Algorithm 2, we can conclude that.

1. Step 8 of the algorithm can be implemented in $O(lg(n))$ time by using a heap implementation.
2. Step 9 will require $O(n)$ time to find the minimum slack task among both *Active* and *Inactive* task lists.
3. Rest of the steps can be completed in $O(1)$ time, as they are merely mathematical computations, based on well defined formulae.

Thus the complexity of single call to *NextShutdownPoint()* is $O(n)$.

Theorem 4. *In worst case, a constant number of calls are made to update slacks per lifetime of a task instance in the system.*

PROOF. In light of the description in Algorithm 1, we can conclude that.

1. Task slacks are updated when either a new task arrives or a task finishes its execution or a shutdown point is reached.
2. For any task instance T_i, this means that once the update of slacks in system is done on its arrival and once when it finishes.
3. A processor shutdown point is computed only when either a task arrives or leaves the system. Thus for any task T_i there can be in worst case 2 calls to compute $t_{shutdown}$, which implies there can be only 2 processor shutdown events.

Thus in total there can be at most 4 calls to update system slack per task instance lifetime.

Theorem 5. *The amortized scheduling complexity C per timeslot of the EPPS algorithm is:*

$$C = \begin{cases} O(m \cdot lg(n)) & ; \quad E \geq \frac{n}{lg(n)} \\ O(\frac{nm}{E}) & ; \quad otherwise. \end{cases} \tag{10}$$

where, n denotes the number of tasks at any given time, m the number of processors and E the average length of time for which a task / application executes on the system.

PROOF. Based on description provided in Algorithm 1, we can say that:

1. For each time slot, steps 3 and 4 take $O(mlg(n))$ time, due to m removals and insertions in *Active* heap.
2. The remaining time requirements are dominated by call to *NextShutdownPoint()* and updating system slacks for the task.
3. As per Theorem 4 the for a single processor lifetime the number of calls to update slack and *NextShutdownPoint()* are constant (say c_i for task T_i). Thus for n tasks the total number of such calls is $\sum_{i=1}^{n} c_i = O(n)$.
4. As per Theorem 3 the complexity of each such call is $O(n)$. Thus for the lifetime of n tasks the total complexity of updating slacks and calling *NextShutdownPoint()* becomes $O(n^2)$.
5. The minimum time required to complete n tasks is $\frac{sum_{i=1}^{n} e_i}{m}$. Let the average execution time requirement of tasks is E. Then the minimum time required to complete the n tasks becomes $\frac{nE}{m}$.
6. Thus maximum time taken by update slacks and *NextShutdownPoint()* per time-slot is given as $O\left(\frac{m \times n^2}{E}\right)$.
7. When $E \geq \frac{n}{lg(n)}$ the complexity per timeslot becomes $O(mlg(n))$.

Note that the complexity derived in Theorem 5 is not a tight upper-bound, since it doesn't take into account the processor shutdown, which will further increase the expected lifetime of a task, thereby reducing the complexity of the scheduler per timeslot.

5. Results

Simulation based experiments were conducted to compare the efficacy of the proposed scheduler with ESPS [5] and Basic-ERFair scheduler. To adapt these schedulers to the multiprocessor environment some modifications were made, as to when the processors are shutdown.

ESPS: This scheduler is primarily used for uniprocessor shutdown. The multiprocessor version used in the experiments shuts down all the processors of the system at a time. Rest of the details of the algorithm were kept same as described in [5].

Basic-ERFair: Basic ERFair algorithm simply takes the m most urgent task and executes them on the available processors. It does not encompass the possibility of processor shutdown. In the experiments done, processors were shutdown when there were no active tasks in the system and the expected time for arrival of the next task instance is at least $T_{breakpoint}$ timeslots away.

5.1. Datasets

The dataset used in the simulations consisted of randomly generated hypothetical task sets. The weights of the tasks were generated using the *randfixedsum* method as described in [21], while the period of the tasks were taken from normal distributions with $\sigma = \mu/5$. Unless stated otherwise, in the following

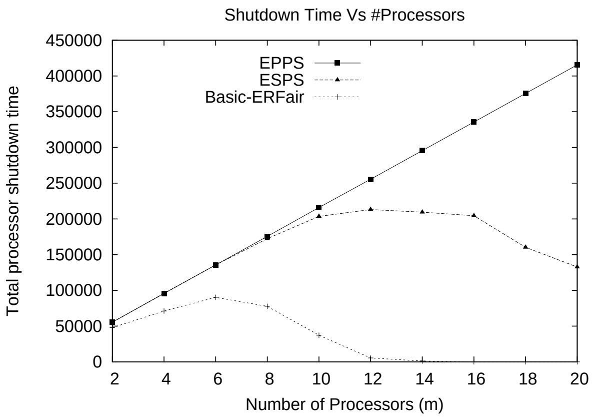

Figure 1: #Tasks(N)=30, System Load (L) = 75%

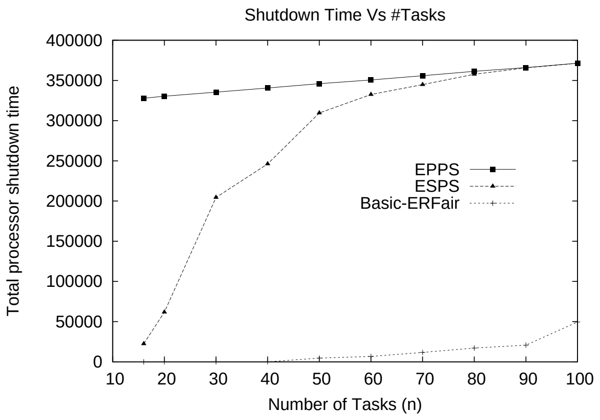

Figure 2: #Processors(m)=16, System Load(L)=75%

sections, the periods were taken from a normal distribution with $\mu = 100$ and $\sigma = 20$. Task sets were generated for number of tasks ranging from 10 to 100 and number of processors varying from 2 to 20. The values system load considered for simulation were 55%, 65%, 75%, 85% and 90%.

5.2. *Simulations*

In the simulations the value of $T_{breakpoint}$ is taken to be $3ms$ (unless stated otherwise), based on the power model of Transmeta Cruseo Processor with $70nm$ technology. This section explores the performance of $EPPS$ algorithm in comparison to $ESPS$ and $Basic - ERFair$ algorithm, while varying system parameters like number of processors (m), system load (L), average task period (P) and $T_{breakpoint}$. All the results in this section are averaged over 100 runs of the scheduler for 100000 time slots of schedule length.

5.2.1. *Effect of Processors*

Figure 1 shows the variation in the shutdown lengths achieved by candidate algorithms, when the number of processors are changed in the system, keeping the other parameters constant.From the plot it can be seen that the for the range considered, the $EPPS$ algorithm shows a linear increase in the total shutdown time as the number of processors are increased. However the $ESPS$ and $Basic - ERFair$ do not scale to the increase in processors. This observation can be reasoned as, since the number of tasks and system load are fixed for the variation in the number of processors, as the number of processors increase, the average task weight increases. Also since the average period of a task in the experiment is 100, it can be said that with increase in processors, the execution requirements of a single task increase. As the weight of some tasks approach 1, all the processors can not be shutdown at the same time (some processors need to serve tasks with high weight). Hence in such cases the time instances when all the processors are free and there is no task in the list, become more scarce. This results in poor performance of $Basic - ERFair$ and $ESPS$ algorithm for higher number of processors.

5.2.2. *Effect of Tasks*

Figure 2 shows the effect of varying number of tasks in system on shutdown time. As reasoned in previous section, the performance of $Basic - ERFair$ and $ESPS$ algorithm is better when the average task weight in the system is low. In this experiment, the number of processors (m), Average task period (P) and the system load (L) are kept constant. This means as the number of tasks increase, the average task weight (and hence the average execution requirement) will decrease. Hence as the number of tasks is increased the $ESPS$ algorithm starts to approach the performance of $EPPS$ algorithm (as evident from Figure 2). The performance of $Basic - ERFair$ scheduler does not increase much because another effect of increase in the number of tasks is that, the system queue does not become empty very often which counteracts on the effect on decrease in task weights, this results in lower increase in performance of $Basic - ERFair$ scheduler than compared to $ESPS$.

5.2.3. *Effect of Period*

Figure 3 shows the effect of increasing the average task period, on the shutdown time achieved by $ESPS$ and $EPPS$ algorithms (the shutdown time for $Basic - ERFair$ scheduler was very low as compared to these two schedulers and thus has been omitted for comparison). In is evident from the plots in Figure 3 that as the average period of tasks increase, the performance of $ESPS$ approaches that of $EPPS$ algorithm. For $EPPS$ algorithm the shutdown length initially increases on increasing the task period and then remains almost constant. The improvement in the results for shutdown interval can be attributed to the fact that, in these experiments the value of $T_{breakpoint}$ is kept constant at 3. As the period of tasks increase, the feasible time interval for shutting down the processors of the system also increases. Hence for high values of task periods, most of the processor

System Load (L)	$P = 50$		$P = 100$		$P = 200$	
	ESPS	*EPPS*	*ESPS*	*EPPS*	*ESPS*	*EPPS*
55	477971.07	670898.98	587655.18	655787.55	608572.76	647752.82
65	265660.03	511016.92	349571.66	495566.49	411371.81	487732.69
75	109954.15	351143.4	204502.468	335550.933	261601.3	327652.18
85	23854.94	191215.64	67530.63	175838.16	113004.28	167621.43
90	12384.68	111946.94	15379.7	95567.76	41036.55	87914.15

Table 1: Shutdown time in timeslots(m = 16, N=30)

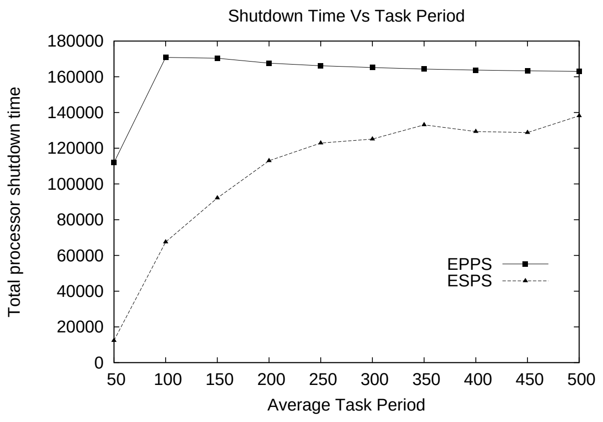

Figure 3: #Processors(m)=16, #Tasks(N)=30, System Load (L)=75 %

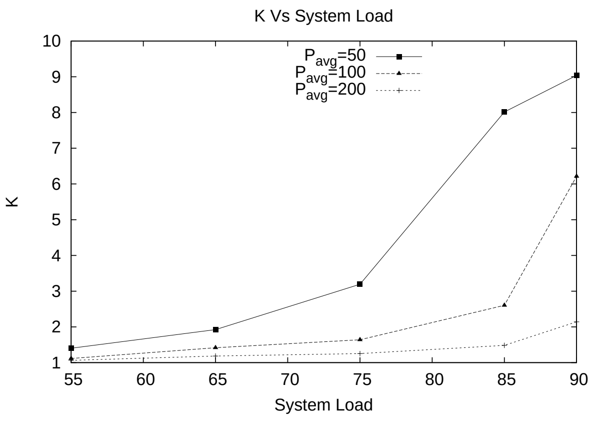

Figure 4: #Processors (m) = 16, #Tasks (N) = 30

shutdown points are valid. This results in the increase of shutdown intervals for *ESPS* algorithm as it gradually approaches the performance of *EPPS* algorithm.

5.2.4. Effect of System Load

Table 1 shows the shut down time for 100000 time slots of schedule on a system with 16 processors and 30 tasks. As expected the shutdown time decreases as the system load is increased. Also as depicted in the previous section, as the average period of the tasks is increased, the total shutdown time also increases for both the algorithms. To compare the performance of *ESPS* and *EPPS* algorithm we define:

$$K = \frac{\text{Shutdown length for } EPPS}{\text{Shutdown length for } ESPS} \tag{11}$$

Figure 4 shows the variation of K for task sets with different periods, as the system load is varied. It can be seen that although as per table 1 the shut down length for both the algorithm decrease as system load increases, the decrease in *ESPS* algorithm is more prominent, since the value of K increases with increase in system load. This observation can be reasoned on the grounds that, as the system load increases, since the number of tasks are constant, the average task weight increases. As already depicted in previous sections, an increase in task weight is detrimental to performance of *ESPS* algorithm, whereas its effect on *EPPS* is not that profound.

5.2.5. Effect of $T_{breakpoint}$

Figure 5 shows the variation in the shutdown time achieved by *ESPS* and *EPPS* algorithms for a system load of 75% with 16 processors and 30 tasks in the systems with an average period of 100. Since *ESPS* algorithm tries to shutdown all the processors at the same time, the shutdown intervals for each individual processor shutdown is very low as compared to the *EPPS* algorithm (which shuts down processors greedily). Thus as the value of $T_{breakpoint}$ is increased, *ESPS* algorithm fails to find proper shutdown points owing to constraint of shutting down all the processors at the same time. While *EPPS* algorithm also shows a slight decrease in the average shutdown time, its negligible as compared to performance degradation of *ESPS* algorithm, which shows an almost linear decrease in the achieved shutdown time as the value of $T_{breakpoint}$ increases.

6. Conclusion

To the best of author's knowledge *EPPS* is the first algorithm for processors shut down in ERFair system with multiple processors. The proposed scheduler performs better than naive adaptations of single processor algorithms like *ESPS*, while not compromising on the algorithm complexity for scheduling. The EPPS algorithm exploits available system slack to shut down processors greedily while locally maximizing the shutdown intervals. The simulation results for the algorithm show a good

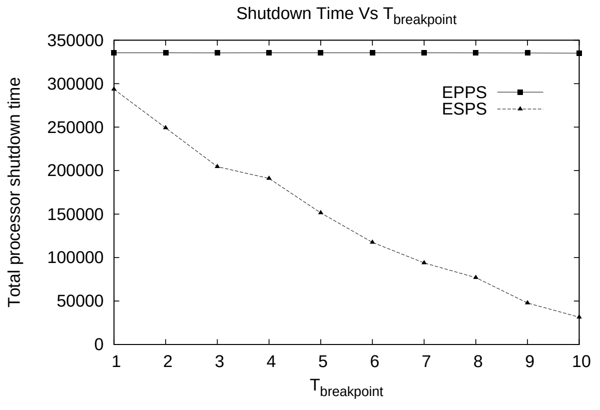

Figure 5: #Processors (*m*) = 16, #Tasks (*N*)= 30, System Load (*L*)=75 %

performance on shutting down processors in real time ERFair systems.

References

[1] W. Yuan and K. Nahrstedt, "Energy-efficient cpu scheduling for multimedia applications," *ACM Trans. Comput. Syst.*, vol. 24, no. 3, pp. 292–331, Aug. 2006. [Online]. Available: http://doi.acm.org/10.1145/1151690.1151693

[2] H. Aydin, R. Melhem, and et al., "Determining optimal processor speeds for periodic real-time tasks with different power characteristics," in *IN PROCEEDINGS OF EUROMICRO CONFERENCE ON REAL-TIME SYSTEMS*, 2001, pp. 225–232.

[3] Y. Shin, K. Choi, and T. Sakurai, "Power optimization of real-time embedded systems on variable speed processors," in *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, ser. ICCAD '00. Piscataway, NJ, USA: IEEE Press, 2000, pp. 365–368. [Online]. Available: http://dl.acm.org/citation.cfm?id=602902.602984

[4] B. Zhai, D. Blaauw, D. Sylvester, and K. Flautner, "Theoretical and practical limits of dynamic voltage scaling," in *Proceedings of the 41st annual Design Automation Conference*, ser. DAC '04. New York, NY, USA: ACM, 2004, pp. 868–873. [Online]. Available: http://doi.acm.org/10.1145/996566.996798

[5] A. Sarkar, S. Swaroop, S. Ghose, and P. Chakrabarti, "Erfair scheduler with processor shutdown," in *High Performance Computing (HiPC), 2009 International Conference on*, dec. 2009, pp. 4 –12.

[6] J. Augustine, S. Irani, and C. Swamy, "Optimal power-down strategies," in *Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on*, oct. 2004, pp. 530 – 539.

[7] J. Anderson and A. Srinivasan, "Early-release fair scheduling," in *12th Euromicro Conference on Real-Time Systems*, Jun 2000, pp. 35–43.

[8] S. Ramabhadran and J. Pasquale, "Stratified round robin: A low complexity packet scheduler with bandwidth fairness and bounded delay," in *ACM SIGCOMM*, 2003, pp. 239–249.

[9] A.-W. H. Stoica, Ion, K. Jeffay, S. Baruah, J. Gehrke, and C. G. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *17th IEEE Real-Time Systems Symposium*, December 1996. [Online]. Available: citeseer.ist.psu.edu/stoica96proportional.html

[10] J. Regehr, M. Jones, and J. Stankovic, "Operating system support for multimedia: The programming model matters, Tech. Rep. MSR-TR-2000-89, Sep 2000.

[11] M. Chrobak and C. Drr, "Polynomial time algorithms for minimum energy scheduling," 908.

[12] M. Bender, R. Clifford, and K. Tsichlas, "Scheduling algorithms for procrastinators," *Journal of Scheduling*, vol. 11, pp. 95–104, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10951-007-0038-4

[13] P. Baptiste, M. Chrobak, and C. Dürr, "Polynomial time algorithms for minimum energy scheduling," *CoRR*, vol. abs/0908.3505, 2009.

[14] J. Chen and T. Kuo, "Procrastination for leakage-aware rate-monotonic scheduling on a dynamic voltage scaling processor," in *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2006, pp. 153–162.

[15] ——, "Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems," in *ICCAD '07: 2007 IEEE/ACM international conference on Computer-aided design*. Piscataway, NJ, USA: IEEE Press, 2007, pp. 289–294.

[16] R. Jejurikar and R. Gupta, "Procrastination scheduling in fixed priority real-time systems," *ACM SIGPLAN Notices*, vol. 39, no. 7, pp. 57–66, 2004.

[17] ——, "Dynamic slack reclamation with procrastination scheduling in real-time embedded systems," in *DAC '05: 42nd annual conference on Design automation*. New York, NY, USA: ACM, 2005, pp. 111–116.

[18] R. Jejurikar, C. Pereira, and R. Gupta, "Leakage aware dynamic voltage scaling for real-time embedded systems," in *DAC '04: 41st annual conference on Design automation*. New York, NY, USA: ACM, 2004, pp. 275–280.

[19] Z. Lu, Y. Zhang, M. Stan, J. Lach, and K. Skadron, "Procrastinating voltage scheduling with discrete frequency sets," in *DATE '06: conference on Design, automation and test in Europe*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 456–461.

[20] J. R. Lorch and A. J. Smith, "Improving dynamic voltage scaling algorithms with pace."

[21] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for synthesis of mutliprocessor tasksets," *WATERS*, pp. 6–11, 2010.