Integrating Processor Slowdown and Preemption Threshold Scheduling for Energy Efficiency in Real Time Embedded Systems

Ravindra Jejurikar Rajesh K. Gupta

Center for Embedded Computer Systems,
Department of Information and Computer Science,
University of California at Irvine,
Irvine, CA 92697
E-mail: `jezz@ics.uci.edu, gupta@cs.ucsd.edu`

CECS Technical Report #04-03
February 2004

Abstract

Preemption threshold scheduling (PTS) enables designing scalable real-time systems. PTS not only decreases the run-time overhead of the system, but can also be used to decreases the number of threads and the memory requirements of the system. In this paper, we combine preemption threshold scheduling with dynamic voltage scaling to enable energy efficient scheduling in real time systems. We consider PTS with task priorities defined by the earliest deadline first policy. We present an algorithm to compute threshold preemption levels for tasks with given static slowdown factors. The proposed algorithm improves upon known algorithm in terms of time complexity. Using our approach, we show that PTS can be used to substantially minimize the context switching overhead even in the presence of slowdown. Experimental results show that preemption threshold scheduling reduces the context switches by 50% - 90%. *Further, we describe a dynamic slack reclamation technique that working in conjunction with PTS and yields further energy gains from 10-50% depending upon availability of slack.*

Contents

List of Figures

1 Introduction

With increasing mobility and proliferation of embedded systems, low power consumption is an important aspect of embedded systems design. Generally speaking, the processor consumes a significant portion of the total energy primarily due to increased computational demands and computing power in these systems. Scaling the processor frequency and voltage based on the performance requirements can lead to considerable energy gains. It is known that preemptability is a necessary requirement to achieve higher processor utilization and optimal slowdown [2, 3]. However, preemptive scheduling has its additional costs as compared to non-preemptive scheduling. Though preemptive scheduling achieves higher utilization, it is not always required to preempt a lower priority task. *Preemption threshold scheduling* [28] [29] allows a task to disable preemptions from tasks up to a specified preemption threshold priority. Tasks with a priority greater than the preemption threshold priority are still allowed to preempt. The preemption threshold scheduling model has been shown to reduce the run-time costs by eliminating unnecessary task preemptions. Furthermore, PTS allows tasks to be partitioned into non-preemptive groups to minimize the number of threads [29] and the stack memory requirement [8], thereby leading to scalable real-time systems.

It is important to minimize context switching taking into account its associated overhead. Context switching typically consists of saving the registers and updating the task control block (TCB) so as to reflect the context switch. In addition, there is an associated overhead of computing the highest priority task in the system and restoring its context when it resumes execution. This context switch overhead is shown to be of the order of 2-30 μsec for Linux and Windows operating system on a personal computer [5]. Though task preemption takes only a few microseconds, the effective time and energy overhead of preemption is larger when considering the energy consumption of components like caches and Translation Look-aside Buffers (TLBs) which rely on the locality of references. Since the execution resumes at a new location on a context switch, it can lead to increased time and energy consumption in these components. Further, if tasks use other resource such as floating point units (FPUs) and other co-processors, the context of these resources must be saved as well.

Caches are an important source of variability in context switching overheads. Based on the processor architecture, a cache miss can result in a delay between 4 to 50 processor cycles [22]. Mogul *et al.* [23] show that the CPI (cycles per instruction) increases by 4%-6% over the $100,000$ instructions after a context switch. The effect of a context switch on cache performance has also been studied in the last decade. Agarwal *et al.* [1] show that the cache miss rate increases with multi-tasking. Tagged caches are used in multi-programming environments to avoid flushing the cache at every context switch. Even with tagged caches, the cache miss rate increases by 30% to 40% due to multi-programming. If the cache is flushed on every context switch, the miss rate increases up to 5 times. Furthermore, when flushing a dirty data block, it needs to be written back to memory. Agarwal *et al.* also show that 30%-50% of the data cache lines are dirty and need to be written back to the memory (or L2 cache). The important point to note is that the energy per memory access increases by 1 to 3 orders of magnitude for each higher level of the memory hierarchy [7], [18]. Hence, cache misses result in a large energy overhead in the memory subsystem, in addition to the time overhead. Filter caches, TCM (tightly coupled memory) and dedicated buffers have been proposed for improved performance, which add to the overhead due to context switching. Translation Lookaside Buffers (TLBs), used for translation of virtual to physical address and Branch Target Buffers (BTBs), used in predicting the target addresses of branch instructions also rely on locality of references, and add to the context switching overhead. Techniques like prefetching [35] and Hit-under-Miss [13] are useful in reducing the time overhead, however the energy overhead does not decrease.

The context switch overhead further escalates as techniques such as Dynamic Voltage Scaling (DVS) and Dynamic Cache Reconfiguration (DCR) are used to decrease the energy consumption. These two techniques rely upon assignment of different slow down factor [2] and different cache configurations [33] that leverage differences in task characteristics to minimize power consumption. This increases the context switch overhead since tasks (applications) require architectural reconfigurations (fine tuning) such as DVS, DCR etc. on a context switch. Thus in addition to the context switch overhead, tasks can have an overhead of changing the processor speed and cache configuration. Though processors like PowerPC 405LP [6] support a voltage change while executing a task, processors like Xscale [13] have an overhead of $20\mu secs$ to change the processor speed. Similarly cache reconfiguration has the overhead of flushing the entire cache and beginning execution with an empty cache. As mentioned earlier, flushing the cache at every context switch can increase the cache miss ratio by up to 5 times. More recently, multi-core architectures [16] have been proposed for energy efficiency, where different cores provide an energy efficient execution based on the time and energy budget. With tasks assigned to different processing cores, the context switch overhead can be as large as a processor shutdown overhead. Thus it is important that the context switches be reduced to the minimum.

Preemption threshold scheduling (PTS) eliminates unnecessary context switches, thereby saving energy. In this paper, we integrate processor slowdown with preemption threshold scheduling to enable scalable and energy efficiency real-time systems. Given task with static slowdown factors, we propose an algorithm to compute the preemption threshold levels for tasks that runs in time $O(n^2)$. This improves upon known algorithms that at best are $O(n^3)$. We show that PTS significantly reduces the number of context switches. We also propose a dynamic slack reclamation scheme that works in conjunction with PTS to minimize the system energy based on run-time conditions.

The rest of the paper is organized as follows: The related work is discussed in Section 2. In Section 3, we present an algorithm to compute the task preemption threshold levels under the EDF priority assignment. A dynamic slack reclamation algorithm that works in conjunction with preemption threshold scheduling is discussed in Section 4. The experimental results are given in Section 5. Finally, Section 6 concludes the paper with future directions.

2 Related Work

Previous work on energy aware scheduling mainly focuses on preemptive scheduling, which is commonly used in processor scheduling. Among the earliest works, Yao *et al.* [31] presented a optimal off-line algorithm to schedule a given set of jobs with arrival times and deadlines. For a similar task model, optimal algorithms have been proposed for fixed priority scheduling [25, 32] and scheduling over a fixed number of voltage levels [17]. The problem of scheduling for real-time task sets for energy efficiency has also been addressed. Real-time schedulability analysis has been used in previous works to compute static slowdown factors for the tasks [27], [10]. Aydin *et al.* [2] address the problem of minimizing energy, considering the task power characteristics. Dynamic slowdown techniques [24], [3], [15] have been proposed to further increase the energy gains at run-time. The problem of maximizing the system value for a specified energy budget, as opposed to minimizing the total energy, is addressed in [26].

Non-preemptive scheduling has been addressed in the context of multi-processor scheduling. Scheduling periodic and aperiodic task graphs which capture the computation and communication in a system is considered in [21],[11]. Zhang *et al.* [34] have given a framework for non-preemptive task scheduling and voltage assignment for dependent tasks on a multi-processor system. They have formulated the voltage scheduling problem as an integer programming problem. The problem of minimizing the energy consumption by performing a slowdown tradeoff in the computation and communication subsystems is addressed in [20], [19]. Previous work on energy aware scheduling considers either preemptive priority scheduling or non-preemptive scheduling.

Preemption threshold scheduling (PTS) for fixed priority systems has been proposed by Wang and Saksena [28], [29]. The authors show that this scheduling model improves schedulability, withholds unnecessary preemptions and reduce the number of threads (processes) in the system thereby leading to scalable system designs. Gai *et al.* [8] extend this scheduling model to the EDF priority assignment and show that it can reduce the memory requirements of the system. Recent works have extended this model to consider scheduling in the presence of task synchronization [14]. Processor slowdown and preemption threshold scheduling have not been addressed and we propose static and dynamic slowdown algorithms that work in conjunction with preemption threshold scheduling.

3 EDF Preemption Threshold Scheduling

Earliest Deadline First (EDF) scheduling is the optimal scheduling policy and can achieve a processor utilization of 1. In this section, we consider preemption threshold scheduling under the EDF priority scheduling. The necessary and sufficient condition for the feasibility of a task set is that the processor utilization be less than or equal to one, $\sum_{i=0}^{n} \frac{C_i}{T_i} \leq 1$. Considering task slowdown factors, the feasibility condition is that the utilization under the given task slowdown factors be at most 1. Given a slowdown factor of η_i for task τ_i,

$$\sum_{i=0}^{n} \frac{1}{\eta_i} \frac{C_i}{T_i} \leq 1 \tag{1}$$

is a necessary and sufficient feasibility test under EDF scheduling.

3.1 Preemption Threshold Scheduling

Preemption threshold scheduling has also been extended to EDF scheduling policy [8], and this analysis can be applied to other optimal dynamic priority scheduling policies. Similar to task priorities in fixed priority systems, task preemption levels can be used for dynamic priority systems. Task preemption levels have been introduced by Baker [4], where a *preemption level*, $\pi(\tau)$, is associated with each task in addition to the task priority. The essential property of the preemption level, $\pi(\tau)$, is that a task τ' is not allowed to preempt another task τ unless $\pi(\tau') > \pi(\tau)$. The preemption level is statically assigned to each job and applies to all execution requests of a job.

Similar to the preemption threshold priority, a threshold preemption level $\gamma(\tau_i)$ is defined for each task. When a task begins execution its preemption level is raised to its threshold preemption level. Under the preemption threshold scheduling, a task τ' preempts a task τ, if task τ' has a higher priority and a higher preemption level than the task τ. Since a task preemption level is raised to its threshold preemption level $\gamma(\tau)$, if τ' preempts τ then.

$$P(\tau') > P(\tau) \; and \; \pi(\tau') > \gamma(\tau) \tag{2}$$

When a higher priority task does not preempt a lower priority task, we say that the higher priority task is blocked. Note that the feasibility of the task set is guaranteed in the presence of this blocking.

3.2 Feasibility Test

We discuss the feasibility of a task set with assigned preemption levels ($\pi(\tau)$) and threshold preemption levels ($\gamma(\tau)$). With the task preemption levels being inversely proportional to the task period, all tasks satisfy the definition of preemption level [4]. Under preemption threshold scheduling, a task can be blocked by lower preemption level tasks that have a higher or equal threshold preemption level and the blocking time, B_i for a task is given by

$$B_i = max_j \left\{ \frac{C_j}{\eta_j} | \pi(\tau_i) > \pi(\tau_j) \ and \ \pi(\tau_i) \leq \gamma(\tau_j) \right\} \tag{3}$$

The computation of B_i for each task takes time linear in the number of task and computing the B_i values for all n tasks takes $O(n^2)$ time. A sufficient condition for the feasibility of a task set, proposed by Baker [4] is given below.

$$\overset{\forall i}{i = 1, ..., n} \quad \frac{B_i}{T_i} + \sum_{k=1}^{i} \frac{1}{\eta_k} \frac{C_k}{T_k} \leq 1 \tag{4}$$

Given the B_i values the feasibility of the task set can be computed in linear time. Since the computation of B_i takes $O(n^2)$ time, the time required to test the feasibility of the task set is $O(n^2)$.

3.3 Computing Threshold Preemption Levels

Given the task preemption levels (priority) for a system with dynamic (fixed) priority assignment, the computation of task threshold preemption levels (preemption threshold priority) is discussed in [28] [8]. A search space of $O(n^2)$ is explored by the algorithms to compute the optimal threshold preemption levels. The algorithms invoke a feasibility test for each of the $O(n^2)$ choices in the search space. The feasibility test discussed above (Equation 4) requires time $O(n^2)$, the worst case time complexity of the algorithms is $O(n^4)$. Note that, based on the order in which the search space is traversed, the B_i values can be incrementally computed in linear time. Thus the feasibility at each element can be computed in linear time, leading to a worst case computation time of $O(n^3)$. We present a faster algorithm which runs in a worst case time of $O(n^2)$ to compute the same solution.

To have a faster algorithm, we compute a worst case blocking time, Y_i, that each task τ_i can tolerate while ensuring all deadlines. The algorithm begins with the tasks sorted in non-decreasing order of their relative deadline, with $i < j$ implying $T_i \leq T_j$. The task set is feasible if the maximum blocking time Y_i for each task τ_i satisfies the feasibility test given by Equation 4,

$$\overset{\forall i}{i = 1, ..., n} \quad \frac{Y_i}{T_i} + \sum_{k=1}^{i} \frac{1}{\eta_k} \frac{C_k}{T_k} \leq 1 \tag{5}$$

We compute the threshold preemption levels using the Y_i values. The algorithm to compute the threshold preemption levels is given in Figure 1. Using Equation 5, we can compute the Y_i values in linear time as seen in the first four lines of the algorithm. A task does not block any other task with the threshold preemption level of a task equal to its preemption level, and the threshold preemption level of each task

Algorithm 1 Computation of Task Preemption Threshold Level, $\gamma(\tau)$

```
1:  for ($i = 1, U_i = 0$; $i \leq n$; $i \leftarrow i + 1$) do
2:      $U_i = U_i + \frac{1}{\eta_i}\frac{C_i}{T_i}$;
3:      $Y_i = (1 - U_i) \cdot T_i$;
4:  end for{End for loop (i);}
5:  for ($i = 1$; $i \leq n$; $i \leftarrow i + 1$) do
6:      $\gamma(\tau_i) \leftarrow \pi(\tau_i)$;
7:      for ($k = i - 1$; $k > 0$ and $Y_k \geq \frac{C_i}{\eta_i}$; $k \leftarrow k - 1$) do
8:          $\gamma(\tau_i) \leftarrow \pi(\tau_k)$;
9:      end for{End for loop (k)}
10: end for{End for loop (i)}
```

is initialized to its preemption level (line 6). The threshold preemption level of the tasks are computed one task at a time. In each step of the algorithm, the threshold preemption level of a task τ_i is increased to the next higher preemption level task τ_k, if the feasible of the task set is maintained. We show that we can incrementally test the feasibility of the task set in constant time. It is known that a task is blocked by at most one task under preemption threshold scheduling [28]. Hence, if Y_k is greater than or equal to C_i/η_i, the execution time of τ_i, then task τ_k meets its deadline even if it is blocked by task τ_i. Thus raising the task threshold preemption level, $\gamma(\tau_i)$ to $\pi(\tau_k)$ maintaining the feasibility of the task set (line 8). Having computed the Y_i values for each task, the feasibility decision can be made in constant time as shown in line 7 of the algorithm. Thus we can traverse the search space of $O(n^2)$ with a constant time feasibility check, leading to a $O(n^2)$ time algorithm. The threshold preemption level of each task cannot be incremented beyond the threshold preemption level computed by the algorithm, since a further increase in the threshold preemption level increases the blocking time and violates the conditions in Equation 4.

4 Dynamic Slack Reclamation

A task usually completes its execution earlier than its worst case number of cycles, resulting in run-time slack. This slack can be used to further reduce the processor speed to result in further energy gains. In this section, we present a slack reclamation scheme for the PTS and is called the Preemption Threshold Scheduling with Dynamic Reclamation (*PTS-DR*) algorithm. With preemption threshold scheduling, higher priority tasks can be present in the ready queue during the execution of a job and we cannot use traditional slack reclamation techniques. We show that tasks can reclaim slack generated by higher or equal priority tasks than the highest priority task in the ready queue, with all tasks meeting the deadline. This is achieved by priority inheritance, wherein if a higher priority task does not preempt the task τ_c being currently processed, the task τ_c inherits the priority (deadline) of the highest priority task in the ready queue. A task can reclaim run-time (slack) with priority higher than or equal to its current priority (based on priority inheritance), while ensuring all task deadlines.

We define *run-time* of a job as the time budget assigned to the job considering its slowdown factor. The run-time of a job with a workload, e, and slowdown η, is e/η. Each run-time also has a priority associated with it, which is set to the job priority. The PTS-DR algorithm is given in Figure 2. The algorithm reserves a run-time for each job based on its static slowdown factor as shown in line 1 of the algorithm. A job consumes run-time as it executes. The unused run-time of jobs is maintained in a priority list called the *Free Run Time list (FRT-list)*. The FRT-list is maintained sorted by priority of the run-times, with the highest priority at the head of the list. Run-time is always consumed from the head of the list. The slack arises due to the early completion of a task and its unused run-time is added to the FRT-list with the same priority as the original priority with which it began execution. A job can use its own run-time as well as the free run-time having a priority no smaller than its priority. If a higher priority job does not preempt a task, then the task inherits the priority of the highest priority ready job. Thus the free run-time that can be reclaimed has a priority no smaller than the jobs in the ready queue.

We use the following notation and definitions in the PTS-DR algorithm as explained below.

- $R_i^r(t)$: the available run-time of the current instance of task τ_i at time t.
- $R_i^F(t)$: the free run-time available to task τ_i at time t. The run-time from the FRT-list with priority $\geq P(\tau_i)$
- $C_i^r(t)$: the residual workload of task τ_i.

The dynamic slowdown factor is the ratio of the residual workload to the available run-time.

The following rules are used in PTS-DR algorithm in consuming the available run-time.

- As task τ_i executes, it consumes run-time at the same speed as the wall clock (physical time). If $R_i^F(t) > 0$, the run-time is used from the FRT-list, else $R_i^r(t)$ is used.
- When the system is idle, it uses the run-time from the FRT-list if the list is non-empty.

Note that the rules need to be applied only on the arrival of a task in the system and on task completion.

With the task deadline used as the priority of the run-time, we show that all tasks meet the deadlines.

Theorem 1 *: All tasks meet the deadline when scheduled by the PTS-DR algorithm.*

The proof is presented in the Appendix A.

4.1 Dynamic Slack Reclamation with Other Implementation Architectures

Preemption threshold scheduling has the advantages of minimizing context switches. In addition, preemption threshold scheduling also enables partitioning the task sets into non-preemption sets. This has shown to reduces the memory requirement in real-time systems [29, 8]. Given static slowdown factors for the tasks, the same algorithm as in previous work can be used to partition the task sets. Dynamic scheduling in the previous section (Section 4) only focused on minimizing context switches. We show that dynamic slowdown can be easily extended to other known implemention archtectures based on preemption threshold scheduling. Dynamic slowdown relies on priority inheritance, which we show can be easily incorporated as described below.

Algorithm 2 Preemption Threshold Scheduling with Dynamic Reclamation (PTS-DR)

```
1: $\{$On arrival of a new task $\tau_i$ :$\}$
2: Let processor be executing $\tau_c$;
3: $R_i^r(t) \leftarrow \frac{C_i}{\eta_i}$;
4: Add task $\tau_i$ to Ready Queue;
5: if ($\tau_c = NULL$) then
6:     return
7: end if
8: if ($P(\tau_i) > P(\tau_c)$ and $\pi_i > \gamma_c$ ) then
9:     Preempt task $\tau_c$;
10: else
11:     if ($P(\tau_i) > P(\tau_c)$) then
12:         Inherit Priority($\tau_c$);
13:         setSpeed $(\frac{E_c^r(t}{R_c^r(t)+R_c^F(t)})$;
14:     end if
15: end if

16: $\{$On execution of each task $\tau_i$ :$\}$
17: setSpeed $(\frac{E_i^r(t)}{R_i^r(t)+R_i^F(t)})$;

18: $\{$On completion of task $\tau_i$ :$\}$
19: Restore priority ($\tau_i$);
20: Add to FRT-list($R_i^r(t), P(\tau_i)$);
```

Gai *et al.* [8] propose using shared virtual resources managed by the stack resource protocol (SRP) [4] to enforce non-preemption among a set of tasks. Priority inheritance is implicit under SRP, since a task needs all resources to be free before it begins execution. Explict priority inheritance can be implemented where a blocking task inherits the priority of the highest priority blocked task. With priority inheritance, tasks can reclaim run-time with a higher or equal priority, while ensuring all task deadlines.

Saksena and Wang [29] dicussed an implemention of preemption threshold scheduling where tasks are mapped to thread to minimize the number of threads and the stack memory requirements. The authors propose an event-based design model consisting of a set of event (types) with computations (actions) associated with each event. Each task (τ_i) is associated with an event E_i and the action associated with the event is the task execution. Each event has a priority, preemption level and a threshold preemption level of the associated task. The assignment of tasks to theads is determined offline, and remains fixed during run-time. Each thread has an associated priority and a preemption level that is used in scheduling threads.

The implementation architecture uses preemptively scheduled threads, where each thread is implemented as an event handler. A thread maintains an event queue where arriving events are queued. The event queue for each thread is maintained as a priority queue, using event priorities. The queued events are processed in a run-to-completion manner, that is, processing of an event is not preempted by the arrival of another event on the thread's event queue. Event processing is done by calling the code associated with the event (action). Whenever a thread selects the next event to process, it is always the highest priority event in the thread event queue. Thread priorities are dynamically managed as follows:

- *On task (event) arrival* : When an event E_i is queued at a thread, then the thread priority is set to the maximum of its current priority and the priority of the event being queued. This takes care of priority inheritance.

- *On beginning of task execution (action)* : When a thread removes an event from its event queue to process, the thread priority is set to the event priority (the highest priority event in the thread) and its preemption level is the maximum of its current thread preemption level and the threshold preemption level of the event.

- *On task (event) completion* : When a thread finishes processing an event, it changes its priority to the highest priority pending event in its event queue. The thread preemption level is also set to the preemption level of this highest priority pending event.

With the thread priority and preemption level managed as above, each thread can reclaim run-time with a higher priority than its current priority, while meeting all task (event) deadlines.

5 Experimental Setup

To evaluate the effectiveness of preemptive threshold scheduling algorithms, we consider several task sets with varying number of synthetically generated tasks. Task periods are generated uniformly in the range [100,1000]. An initial utilization u_i of each task was uniformly assigned in the range [0.05, 0.5]. The Worst Case Execution Times (WCET) for each task was set to $u_i \cdot T_i$ at the maximum processor speed. The task execution times were scaled to ensure a processor utilization less than one, thereby making the task set feasible. Experiments were performed on various task sets and the average results are presented.

We use the power model for dynamic power consumption of CMOS circuits [30]. The dynamic power consumption, P, depends on the operating voltage and frequency of the processor and is given by:

$$P = C_{eff} \cdot V_{dd}^2 \cdot f \tag{6}$$

where C_{eff} is the effective switching capacitance, V_{dd} is the supply voltage and f is the operating frequency. Equation 6 shows the quadratic relationship between power and voltage. However, the transistor gate delay (and hence frequency) depends on the voltage and a decrease in voltage has to be accompanied by a decrease in processor frequency. There is a linear dependence between the frequency and voltage [30], resulting in a linear increases in the execution time of a task. Due to the quadratic decrease in power with voltage, and only a linear decrease in frequency, the energy consumption per unit work decreases with voltage at the cost of increased execution time. The operating voltage range for the processor is $0.6V$ and $1.8V$, which is the trend in current embedded processors [12, 13]. We have normalized the operating speed and support discrete slowdown factors in steps of 0.1 in the normalized range.

5.1 Static Slowdown Factors

Processor slowdown based on static slowdown factors is well studied and known to have significant energy savings. Given a feasible task set with static slowdown factors, we compute threshold preemption levels for the tasks. We assume that the processor utilization of the task-set under maximum speed is used as the static slowdown factor. We compare the scheduling overheads of the following techniques:

- Preemptive Scheduling (PS)
- Preemption Threshold Scheduling (PTS)

Since the same static slowdown factors are used with both preemptive scheduling and preemptive threshold scheduling, the processor energy consumption is the same if the context switch overhead is ignored. Context switching results in additional time and energy overhead and it is beneficial to minimize this overhead. Figure 1 compares the number of context switches under preemption threshold scheduling normalized to the preemptive scheduling policy. The context switches for the various task-sets are shown with the number of tasks along the X-axis and the normalized context switches along the Y-axis. The execution time for each task is its WCET at maximum speed. It is seen that PTS decreases the context switching considerably, with the context switching dropping to 50% for smaller number of tasks and to less than 10% as the number of tasks increase. With smaller number of tasks, there are inherently few context switches and the relative decrease is comparatively low. As the number of tasks increases, we have more tasks with smaller execution times for each task. Such task sets approach closer to non-preemptive scheduling and require very few context switches under preemption threshold scheduling. Under preemptive scheduling, the context switching increases with the number of tasks and PTS performance improves.

We note that context switching results in excess energy consumption especially in the cache and memory subsystem. However, a change in the locality of references is observed not only on a context switch but also when a task resumes execution on completion of another task. A change in the execution

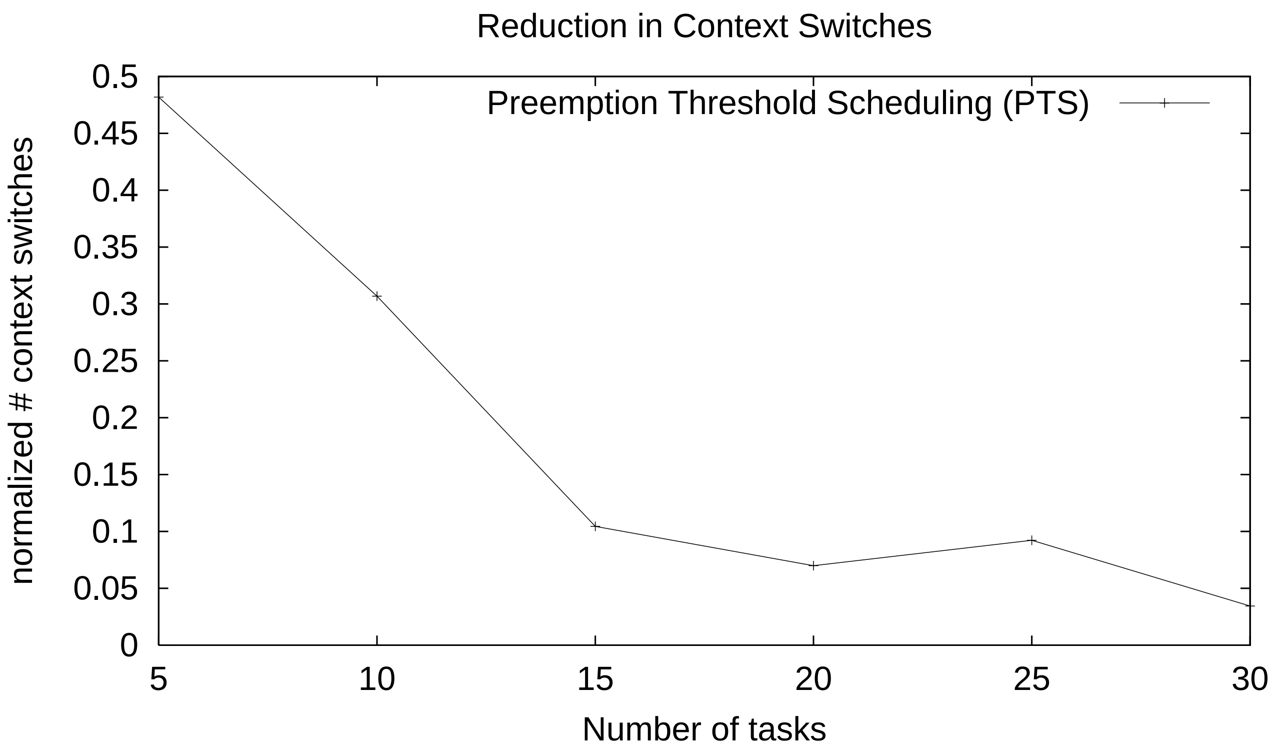


Figure 1. Number of context switches under preemption threshold scheduling normalized to preemptive scheduling.

context is a cause of overhead, and is referred to as an *effective context switch*. Though the overhead of beginning a task execution is comparatively smaller than a context switch, a percentage decrease in the effective context switching gives a good measure of the gains achieved by preemption threshold scheduling. As shown by Agarwal *et al.* [1], the cache performance degrades with multi-tasking with an average cache miss ratio increasing by 30%-40%. These extra cache misses result in more external memory references and an increase in energy consumption. Averaging the cache miss ratio over all effective context switches, a proportionate decrease in the memory energy consumption will be seen. Figure 2 shows the normalized effective context switching between the two scheduling techniques. The graph has a similar pattern as the context switch reducions seen in Figure 1, however note that the context switches reduce by up to 90% whereas the effective context switches reduce by only 25%. We see on an average 25% decrease in the effective context switching which translates to a 25% reduction in the memory subsystem energy consumption. With the processor and memory subsystems consuming comparable energy, this leads to additional energy savings. Preemptive scheduling also has an additional overhead due to the intermittent memory accesses, which reduce the chances of switching the memory to a low power mode. Note that the energy contribution due to context switching is increasing with the fine tuning of application where a context switch can require a voltage change, cache reconfiguration, core reconfiguration and similar architectural reconfigurations which further increase the energy overhead. Thus we see that preemptive threshold scheduling is increasingly important for energy efficient execution of real-time systems.

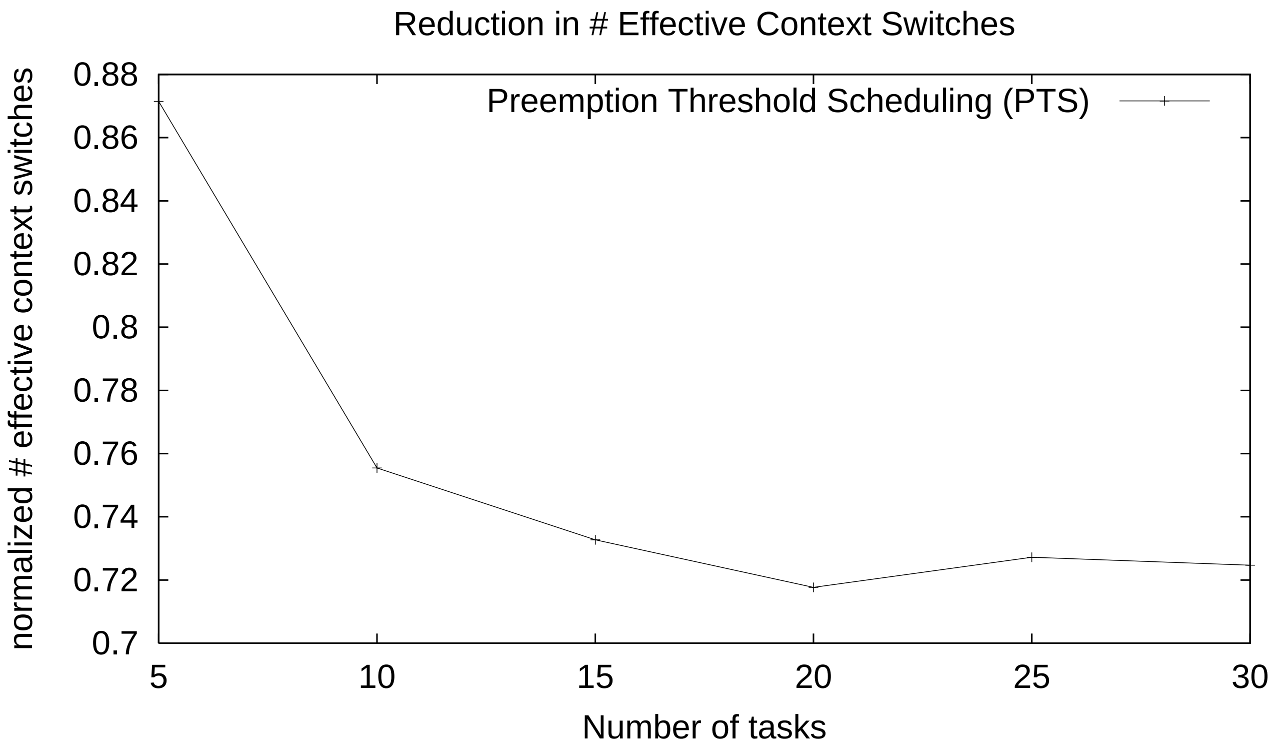


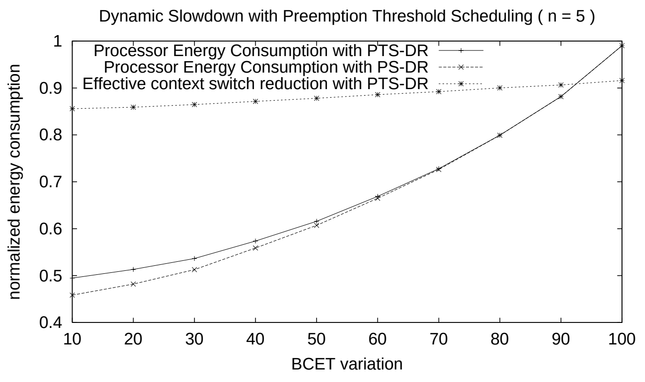
Figure 2. Reduction in the effective context switches under preemption threshold scheduling normalized to preemptive scheduling. Effective context switches reduction is a measure of the energy savings achieved by preemption threshold scheduling (PTS).

5.2 Dynamic Slack Reclamation

We present the energy gains achieved by dynamic slack reclamation. To generate varying execution times, we vary the *best case execution time (BCET)* of a task as a percentage of its WCET. The execution times are generated by a Gaussian distribution with mean, $\mu =$ (WCET+BCET)/2 and a standard deviation, $\sigma =$ (WCET-BCET)/6. The BCET of the task is varied from 100% to 10% in steps of 10%. Experiments were performed on task sets with varying number of tasks (n) per task-set. Figure 3 shows the energy gains for different values of BCET with the number of tasks being $n = 5$ and $n = 25$. Similar results are seen for different values of n. We compare the energy consumption of the following techniques :

- Preemptive Scheduling with Dynamic Reclamation (PS-DR)
- Preemption Threshold Scheduling with Dynamic Reclamation (PTS-DR)

Since task slowdown factors are mapped to discrete voltage levels, there can be dynamic slack in the system even at worst case execution times. Thus dynamic slack reclamation results in energy gains even at BCET of 100%. A steady decrease in the processor energy consumption is seen with a decrease in the BCET. As the task execution time is decreased, there is more slack which results in decreasing the



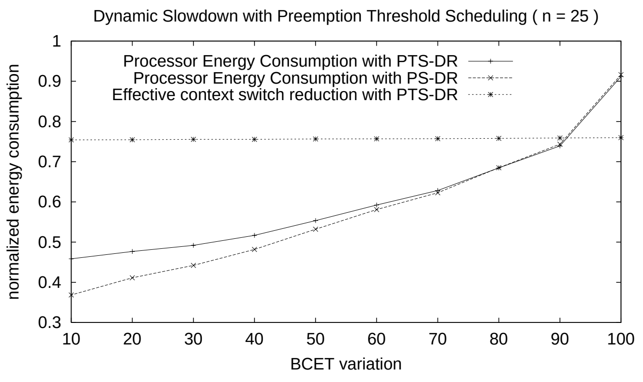


Figure 3. Energy Consumption and reduction in effective context switching with Dynamic Slack Reclamation for (a) number of tasks, $n = 5$ and (b) number of tasks, $n = 25$.

energy consumption by operating at lower voltages. Figure 3 compares the processor energy consumption with PS-DR and PTS-DR. It is seen that the energy consumption with PS-DR and PTS-DR is very close, however the PTS-DR scheme can comsume more processor energy while minimizing the context switches. The energy consumption is similar up to BCET of 60%. With a further reduction in BCET, PTS-DR leads to 4%-8% higher energy consumption than PS-DR. This is due to the fact that PS-DR can reclaim all the higher priority slack at all times. On the other hand, a task inherits the priority when a higher priority task is blocked for its completion, thereby reducing the slack that can be reclaimed. Furthermore, all lower priority task are always preempted under PS-DR, and the preempted task can reclaim the slack generated by the earlier completion of the preempting higher priority task. However, the context switching overhead is very high under preemptive scheduling. Figure 3 also shows the reduction in the effective context switches. PTS-DR gains around 15%-25% reduction in effective context switching, which can directly map to a reduction in the energy consumption of the memory sub-system.

Figure 3 shows the energy consumption for task sets with 5 and 25 tasks. With fewer tasks, it is seen that the difference in the processor energy consumption of PTS-DR and PS-DR is less than 3%. However a reduction in 15% effective context switches is observed. For $n = 25$, PTS-DR leads to processor energy increase of 4%-8%, however reduces the context switching by 25%. Furthermore, the time delay incurred due to excess cache misses in PS-DR will reduce the dynamic slack generated in PS-DR and the energy gains. Gopalakrishnan [9] show that the processor utilization drops as much as 8% with exececssive context switching. This will take away the gains achieved by a PR-DR and thus it is important to minimize the context swithces. PTS-DR reduces the overheads and the energy consumption of the memory subsystem. With the memory subsystem consumption is a significant portion of the total energy, these savings will lead to energy gains over PS-DR. With the increasing time and energy overhead associated with an effective context switch, the total energy gains will keep increasing.

6 Summary and Future Work

We have presented algorithms that help integrate task slowdown and preemption threshold scheduling. Preemption threshold scheduling is important in reducing the context switching among tasks as well as memory requirements of a system. We enable scalable energy efficient real-time systems by integrating preemption threshold scheduling with static and dynamic slowdown. Slowdown decreases the energy consumption of the system and preemption threshold scheduling further reduces the time and energy overheads associated with context switching.

We propose a faster algorithm to compute threshold preemption levels of tasks. Experimental results show on an average 50%-90% decrease in the context switching overhead due to preemption threshold scheduling. We also present a dynamic slack reclamation algorithm that works in conjunction with preemption threshold scheduling. Dynamic slack reclamation can result in 10%-50% further decrease in the energy consumption. These techniques are energy efficient and easy to implement in real systems. These scheduling techniques will increase the energy efficient of systems and will have a great impact on the energy utilization of portable devices. We plan to extend these scheduling techniques with detailed knowledge of task context switching overhead so as to further minimize this overhead.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, 1988.

[2] H. Aydin, R. Melhem, D. Mossé, and P. M. Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *Proceedings of EuroMicro Conference on Real-Time Systems*, 2001.

[3] H. Aydin, R. Melhem, D. Mossé, and P. M. Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, December 2001.

[4] T. P. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[5] E. G. Bradford. Runtime: Context switching. http://www-106.ibm.com/developerworks/linux/library/l-rt9.

[6] B. Brock and K. Rajamani. Dynamic power management for embedded systems. In *Proc. of the IEEE Int'l SOC Conference, Portland, Oregon*, Sept. 2003.

[7] R. Fromm, S. Perissakis, N. Cardwell, C. E. Kozyrakis, B. McGaughy, D. A. Patterson, T. E. Anderson, and K. A. Yelick. The energy efficiency of IRAM architectures. In *Proceedings of International Symposium on Computer Architecture*, pages 327–337, 1997.

[8] P. Gai, G. Lipari, and M. di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of IEEE Real-Time Systems Symposium*, December 2001.

[9] R. Gopalakrishnan and G. M. Parulkar. Bringing real-time scheduling theory and practice closer for multimedia computing. In *ACM SIGMETRICS Conference (Philadelphia, PA)*, pages 1–12, May 1996.

[10] F. Gruian. Hard real-time scheduling for low-energy using stochastic data and dvs processors. In *Proceedings of International Symposium on Low Power Electronics and Design*, pages 46–51, 2001.

[11] F. Gruian and K. Kuchcinski. LEneS: task scheduling for low-energy systems using variable supply voltage processors. In *Proceedings of the Asia South Pacific Design Automation Conference*, 2001.

[12] IBM 405LP Processor. IBM Inc. *(http://www-3.ibm.com/chips/products/powerpc/cores.*

[13] Intel XScale Processor. Intel Inc. *(http://developer.intel.com/design/intelxscale)*.

[14] S. Kim, S. Hong, and T. Kim. Integrating real-time synchronization schemes into preemption threshold scheduling. In *Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 145–152, 2002.

[15] W. Kim, J. Kim, and S. L. Min. A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis. In *Proceedings of Design Automation and Test in Europe*, 2002.

[16] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of International Symposium on MicroArchitecture*, Dec. 2003.

[17] W. Kwon and T. Kim. Optimal voltage allocation techniques for dynamically variable voltage processors. In *Proceedings of the Design Automation Conference*, pages 125–130, 2003.

[18] H. G. Lee and N. Chang. Energy-aware memory allocation in heterogeneous non-volatile memory systems. In *ISLPED*, pages 420–423, 2003.

[19] J. Liu and P. H. Chou. Energy optimization of distributed embedded processors by combined data compression and functional partitioning. In *Proceedings of International Conference on Computer Aided Design*, Nov. 2003.

[20] J. Liu, P. H. Chou, and N. Bagherzadeh. Communication speed selection for embedded systems with networked voltage-scalable processors. In *Proceedings pf International Symposium on Hardware/Software Codesign*, Nov. 2002.

[21] J. Luo and N. Jha. Power-conscious joint scheduling of periodic task graphs and a periodic tasks in distributed real-time embedded systems. In *Proceedings of International Conference on Computer Aided Design*, 2000.

[22] S. Manegold and P. Boncz. Cache-memory and tlb calibration tool. http://homepages.cwi.nl/ manegold/Calibrator/DB/.

[23] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 75–84. ACM Press, 1991.

[24] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of 18th Symposium on Operating Systems Principles*, 2001.

[25] G. Quan and X. Hu. Minimum energy fixed-priority scheduling for variable voltage processors. In *Proceedings of Design Automation and Test in Europe*, March 2002.

[26] C. Rusu, R. Melhem, and D. Mosse. Maximizing rewards for real-time applications with energy constraints. In *ACM Transactions on Embedded Computer Systems*, accepted.

[27] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Proceedings of International Conference on Computer Aided Design*, pages 365–368, 2000.

[28] Y. Wang and M. Saksena. Scheduling fixed priority tasks with preemption threshold. In *Proceedings of IEEE International Conference on Real-Time Computing Systems and Applications*, Dec 1999.

[29] Y. Wang and M. Saksena. Scalable multi-tasking using preemption threshold scheduling. In *Proceedings of IEEE Real-Time Technology and Applications Symposium*, Jun 2000.

[30] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison Wesley, 1993.

[31] F. Yao, A. J. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, pages 374–382, 1995.

[32] H. Yun and J. Kim. On energy-optimal voltage scheduling for fixed-priority hard real-time systems. *Trans. on Embedded Computing Sys.*, 2(3):393–430, 2003.

[33] C. Zhang, F. Vahid, and W. Najjar. A highly configurable cache architecture for embedded systems. In *Proceedings of International Symposium on Computer Architecture*, pages 136–146, 2003.

[34] Y. Zhang, X. S. Hu, and D. Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proceedings of the Design Automation Conference*, 2002.

[35] D. Zucker, R. Lee, and M. Flynn. Hardware and software cache prefetching techniques for MPEG benchmarks. *IEEE Transactions on Circuits and Systems for Video Technology*, 10(5), 2000.

A Proofs

Lemma 1: The run-time of every task instance is depleted at or before its deadline.

Proof: Suppose the claim is false. Let t be the first time that a run-time of a job or that in a FRT-list is not depleted by its deadline. Let t' be the the latest time before t such that the following two conditions are satisfied: (1) there are no pending jobs with arrival times before t' and deadlines less than or equal to t. (2) The FRT-list does not contain any run-time with deadline less than or equal to t. Since no requests can arrive before system start time $(time = 0)$, t' is well defined. By definition of t', a job τ_h with deadline less than or equal to t arrives at time t' when the system is either idle or executing a job, τ_b, with deadline greater than t. We consider the following two cases depending on whether τ_h begins execution at time t':

- Case I, where τ_h begins execution at time t'. This occurs if the system is either idle before time t' or $\gamma(\tau_b) < \pi(\tau_h)$, which allows task τ_h to preempt task τ_b. In this case, the only run time consumed in the interval $[t', t]$ is that generated by the jobs arriving in the interval $[t', t]$ with deadlines less than or equal to t. Let us denote this run-time by A. Let $X = t - t'$, then the tasks contributing to the run-time in A are a sub-set of the task set $\{\tau_1, \ldots \tau_i\}$ where i is the maximum task index such that $T_i \leq X$. Thus the run-time generated in the interval $[t', t]$, which is denoted by A, is bounded by $\sum_{k=1}^{i} \lfloor \frac{X}{T_k} \rfloor \frac{C_k}{\eta_k}$.
- Case II, where task τ_h is blocked at time t' due to preempting threshold scheduling. Note that if the preemption threshold level of τ_b is greater than or equal to preemption level of τ_h $(\gamma(\tau_b) \geq \pi(\tau_h))$, then task τ_b is not preempted. In this case, run-time apart from A can be consumed. Let the run-time with a deadline greater than t be denoted by B. By definition, it can be seen that the run-times in A and B are disjoint. Note that the run-time in B is only consumed during the execution of τ_b. After the completion of task τ_b, run-time with deadline $\leq t$ is always present for the remaining duration up to interval $[t', t]$ and only the run-time in A is consumed. Thus the run-time in B is bounded by the execution time of task τ_b, that is $\frac{C_b}{\eta_b}$. Since Algorithm 1 assigns task τ_b a preemption threshold level greater than the preemption level of task τ_h $(\gamma(\tau_b) \geq \pi(\tau_h))$, it must he true that $\forall_{h \leq k < b} : \frac{C_b}{\eta_b} \leq Y_k$. Since $h \leq i < b$, it is true that $\frac{C_b}{\eta_b} \leq Y_i$. Thus the run-time in B, consumed by task τ_b, is bounded by Y_i.

In either case, the maximum run-time that can be consumed in $[t', t]$ is bounded by the sum of the run-times in A and B. Since the run-time is not depleted at time t, the sum of the run-time in A and B must be greater than the run-time consumed in the interval $[t', t]$, which is X.

Therefore,

$$Y_i + \sum_{k=1}^{i} \lfloor \frac{X}{T_k} \rfloor \frac{C_k}{\eta_k} > X$$

Since $\frac{X}{T_k} \geq \lfloor \frac{X}{T_k} \rfloor$, we have

$$\frac{Y_i}{X} + \sum_{k=1}^{i} \frac{1}{\eta_k} \frac{C_k}{T_k} > 1$$

Since all jobs that contribute to the run-time in A have their arrival time and deadline in the interval $[t', t]$, we have $T_i \leq X$, and

$$\frac{Y_i}{T_i} + \sum_{k=1}^{i} \frac{1}{\eta_k} \frac{C_k}{T_k} > 1$$

which contradicts with Equation 5. Thus the run-time of every task instance is depleted by its deadline. ■

Theorem 2: All tasks meet the deadline when scheduled by the PTS-DR algorithm.

Proof: Since the task deadlines and the run-time deadlines are the same, the claim follows directly from Lemma 1. Thus all tasks meet the deadline by the PTS-DR algorithm. ■