Integrating Preemption Threshold Scheduling and Dynamic Voltage Scaling for Energy Efficient Real-Time Systems

Ravindra Jejurikar[1] and Rajesh Gupta[2]

[1] Centre for Embedded Computer Systems,
University of California Irvine, Irvine CA 92697, USA
jezz@cecs.uci.edu
[2] Department of Computer Science and Engineering,
University of California San Diego, La Jolla, CA 92093, USA
gupta@cs.ucsd.edu

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 decrease 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 scheduling with task priorities defined by the Earliest Deadline First (EDF) policy. We present an algorithm to compute threshold preemption levels for tasks with given static slowdown factors. The proposed algorithm improves upon known algorithms in terms of time complexity. Experimental results show that preemption threshold scheduling reduces on an average 90% context switches, even in the presence of task slowdown. Further, we describe a dynamic slack reclamation technique that working in conjunction with PTS that yields on an average 10% additional energy savings.

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. Scaling the processor frequency and voltage based on the performance requirements can lead to considerable energy savings. It is known that preemptability is a necessary requirement to achieve higher processor utilization and optimal processor slowdown [1, 2]. However, preemptive scheduling has its additional costs compared to non-preemptive scheduling. Note that it is not always required to preempt a lower priority task while still achieving the same processor utilization as that with preemptive scheduling. Preemption Threshold Scheduling (PTS) [3, 4] allows a task to disable preemption 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 preemption. Furthermore, PTS allows tasks

to be partitioned into non-preemptive groups to minimize the number of threads [4] and the stack memory requirement [5], thereby leading to scalable real-time systems.

It is important to minimize context switching due to 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. Further, if tasks use resource such as floating point units (FPUs) and other co-processors, the context of these resources must be saved as well. 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 [6]. Though task preemption takes only a few microseconds, the effective time and energy overhead of preemption further increases due to components such as 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, preemption can lead to increased execution time and additional energy consumption in these components.

Caches are an important source of variability in context switching overheads. Based on the system architecture, a cache miss can result in a delay between 4 to 50 processor cycles [7]. Mogul et al. [8] 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. [9] 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, flushing a dirty data block requires that the data block 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 [10, 11]. Hence, a higher cache miss rate 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. Note that techniques such as pre-fetching [12] 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 slowdown factor [1] and different cache configurations [14] 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 [15] support a voltage change while executing a task, processors like Xscale [13] have an overhead of 20μsecs 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 as much as possible.

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 even in the presence of task slowdown. We also propose a dynamic slack reclamation scheme that works in conjunction with PTS to minimize the system energy based on run-time conditions. We show that our scheduling techniques result in an energy efficient operation of the system.

The rest of the paper is organized as follows: The related work is discussed in Sect. 2 and Sect. 3 introduces the real-time system model. In Sect. 4, 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 Sect. 5. The experimental results are given in Sect. 6. Finally, Sect. 7 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. [17] 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 [18, 19] and scheduling over a fixed number of voltage levels [20]. The problem of scheduling for real-time periodic task sets for energy efficiency has also been addressed. Real-time feasibility analysis has been used in previous works to compute static slowdown factors for the tasks [21, 22]. Aydin et al. [1] address the problem of minimizing energy, considering the task power characteristics. In our earlier work [23], we have addressed the problem of computing task slowdown factors when task deadlines differ from the task period. Recent works have proposed energy efficient scheduling techniques taking in to account leakage power consumption, which is rapidly increasing due to the exponential increase in the leakage current with each technology generation [24, 25]. Dynamic slowdown techniques [2, 26–28] have been proposed to further increase the energy gains at run-time. The problem of maximiz-

ing the system value for a specified energy budget, as opposed to minimizing the total energy, is addressed in [29].

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 [30, 31]. Zhang et al. [32] 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 [33, 34]. Note that prior works on energy aware scheduling consider either preemptive priority scheduling or non-preemptive scheduling.

Preemption threshold scheduling (PTS) for fixed priority systems has been proposed by Wang and Saksena [3, 4]. The authors show that this scheduling model improves schedulability, withholds unnecessary preemption and reduce the number of threads (processes) in the system thereby leading to scalable system designs. Gai et al. [5] 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 [35]. A combination of processor slowdown and preemption threshold scheduling has not been addressed and we propose static and dynamic slowdown algorithms that work in conjunction with preemption threshold scheduling.

3 Preliminaries

In this section, we introduce the necessary notation and formulate the problem. We first describe the system model and the basics of preemption threshold scheduling.

3.1 System Model

A task set of n periodic real time tasks is represented as $\Gamma = \{\tau_1, ..., \tau_n\}$. A 3-tuple $\{T_i, D_i, C_i\}$ is used to represent each task τ_i, where T_i is the period of the task, D_i is the relative deadline, and C_i is the worst case execution time (WCET) of the task at maximum processor speed. Each invocation of the task is called a *job*. We use the notation of a task and the task instance (job) interchangeably, when the meaning is clear from the context. A priority function $P(\tau)$ is associated with each invocation of a task such that if a task τ has a higher priority than τ', then $P(\tau) > P(\tau')$. With the earliest deadline first priority assignment, the smaller the task deadline the higher the task priority. Note that each task instance has a different priority and $P(\tau)$ represents the priority of the current instance of the task. All tasks are assumed to be preemptive in nature with the relative task deadline equal to the task period ($D_i = T_i$). We say that an executing task is *blocking* another task, if a higher priority task is waiting in the ready-queue for the completion of the current task execution. Such a task would have been preempted under a preemptive scheduling policy, however such tasks exist under preemption threshold scheduling. A task set is said to be *feasible* if all tasks

meet the deadline. The processor utilization for the task set, $U = \sum_{i=1}^{n} C_i/T_i \leq 1$ is a necessary condition for the feasibility of any schedule [36]. Under preemptive threshold scheduling, a preemption level $\pi(\tau_i)$ and a threshold preemption level $\gamma(\tau_i)$ is associated with each task τ_i.

3.2 Preemption Threshold Scheduling

Wang and Saksena [3] have introduced preemption threshold scheduling for fixed priority systems. Under preemption threshold scheduling, each task has a fixed priority and a preemption threshold priority. When a task begins execution its priority is raised to its preemption threshold priority. Thus only tasks with a priority greater than the preemption threshold priority of a task can preempt it. Preemption threshold scheduling enables partitioning a task set into mutually non-preemptive task sets. Since one thread suffices for the implementation of each non-preemptive group of tasks, this reduces the number of threads as well as the stack space requirement of the system.

3.3 Variable Speed Processors

A wide range of processors like the Intel XScale [13], Transmeta Crusoe [37] and PowerPC 405LP [38] support variable voltage and frequency levels. Voltage and frequency levels are tightly coupled. When we change the speed of a processor we change its operating frequency. We proportionately change the voltage to a value which is supported at that operating frequency. The important point to note is when we perform a slowdown we change both the frequency and voltage of the processor. The *slowdown factor* can be viewed as the normalized frequency. At a given instance, it is the ratio of the scheduled frequency to the maximum frequency of the processor. Given the minimum frequency f_{min} and the maximum supported frequency f_{max}, we normalize the speed to the maximum frequency to have discrete points in the operating range $[\eta_{min}, 1]$, where $\eta_{min} = f_{min}/f_{max}$. Processors like the PowerPC 405LP support a voltage change without interrupting the execution of a task. However processors like the Intel XScale, the associated overhead of changing the processor speed is 20μsecs.

The system utilization given a slowdown of η_i for task τ_i is $U_\eta = \sum_{i=1}^{n} \frac{1}{\eta_i} \frac{C_i}{T_i}$. We assume that the overhead incurred in changing the processor speed is incorporated in the task execution time. Considering static and dynamic slowdown, a speed change occurs only when the task begins execution or when a higher priority task is blocked. This overhead is constant and can be incorporated in the worst case processing time of a task.

4 Preemption Threshold Scheduling and Dynamic Priority Systems

In this section, we consider preemption threshold scheduling for dynamic priority systems and consider the Earliest Deadline First (EDF) scheduling policy. EDF scheduling is the optimal scheduling policy and can achieve a processor utilization of 1. 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.

4.1 Preemption Threshold Scheduling

Preemption threshold scheduling has also been extended to EDF scheduling policy [5], 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 [39], 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 under fixed priority scheduling, a threshold preemption level $\gamma(\tau_i)$ is defined for each task under the EDF (dynamic priority) system. 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.

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

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 has to be guaranteed in the presence of this blocking.

4.2 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 [39]. 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

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

Note that the task execution time under slowdown is considered while computing task blocking times. 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 [39] is given below.

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

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 $O(n^2)$.

4.3 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 [3] [5]. 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 presented in (4) requires time $O(n^2)$. However, 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 period, 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 (4),

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

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)}

We compute the threshold preemption levels using the Y_i values. Algorithm 1 describes the computation of task threshold preemption levels. Based on (5), we can compute the Y_i values in linear time as seen in the lines 1-4 of Algorithm 1. A task does

not block any other task when the threshold preemption level of a task is equal to its preemption level, and the threshold preemption level of each task 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 feasibility 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 [3]. 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 $\pi(\tau_k)$ maintains 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 (4).

5 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 additional energy savings. We incorporate dynamic slack reclamation with preemption threshold scheduling.

5.1 Motivation

We show, using an example, that traditional slack reclamation techniques [2, 28] for preemptive task systems can result in a deadline miss under PTS. Consider a task system with the following three tasks

$$\tau_1 = \{5, 10, 10\}, \tau_2 = \{5, 20, 20\}, \tau_3 = \{5, 20, 20\}$$

with task arrival times being $a_1 = 1$, and $a_2 = a_3 = 0$ respectively, as shown in Fig. 1(a). Based on Algorithm 1, it is seen that the threshold preemption level of all tasks is the highest preemption level in the system and hence no task preempts another task. At time $t = 0$, task τ_2 is the highest priority ready task and is scheduled for execution. Assume that it completes execution in 0.5 time units, leaving a slack of 4.5 time units. Task τ_3 is ready at time t=0.5 and begins execution. Traditional slack reclamation schemes enable a task to reclaim slack generated by higher and equal priority tasks [2], which enables task τ_3 to reclaim the slack time of task τ_2. Thus task τ_3 uses the available slack of 4.5 time units along with its own time budget of 5 time units (total budget of 5 + 4.5 = 9.5) to complete execution at $t = 10$. Though task τ_1 arrives at time $t = 1$, it does not preempt task τ_3 since it has the same preemption level as the threshold preemption level of task τ_3. Task τ_1 begins execution at $t = 10$ to finish its execution at time $t = 15$ and

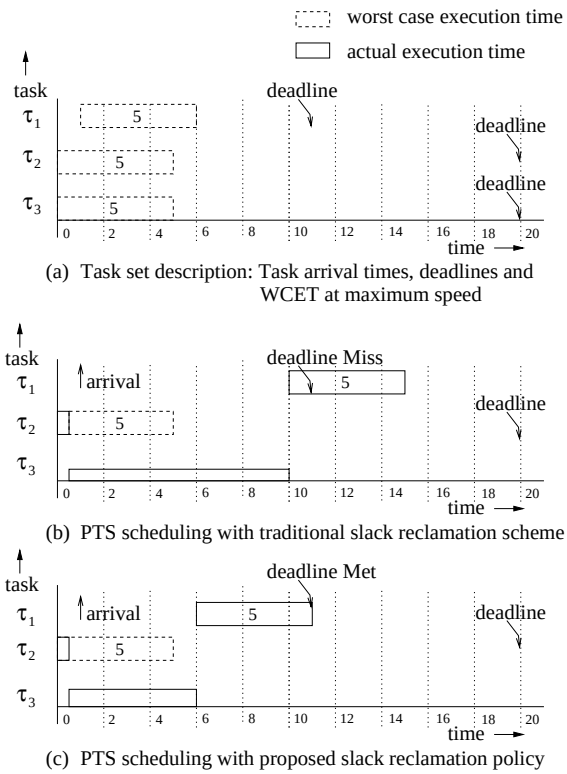

Fig. 1. (a) Task arrival times and deadlines (NOT a task schedule). (b) Traditional slack reclamation with PTS, and task τ_1 misses its deadline. (c) Feasible schedule with proposed slack reclamation policy.

misses its deadline of $t = 11$. This is shown in Fig. 1(b). Thus we see that it is important to determine the slack that can be reclaimed while ensuring all task deadlines.

We present slack reclamation rules that guarantee all task deadlines. 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. Based on these rules, the task schedule is as shown in Fig. 1(c). As shown, task τ_2 finishes at time $t = 0.5$, leaving a slack of 4.5 time units. Task τ_3 begins execution at time $t = 0.5$ and can reclaim this slack up to time $t = 1$ when task τ_1 arrives. At time $t = 1$, task τ_3 inherits the priority (deadline) of the higher priority ready task, τ_1. Since the slack is generated by a task with deadline $d = 20$ and the task has a inherited deadline of $d = 10$, the slack cannot be used by task τ_3. It uses its own time budget of 5 time units to complete at time $t = 6$. Task τ_1 begins execution at $t = 6$ to complete at time $t = 11$ and meets its deadline. We later prove that a task can

reclaim slack (run-time) generated by higher or equal priority task (based on priority inheritance), while ensuring all task deadlines.

5.2 Slack Reclamation Algorithm

We now describe in detail the slack reclamation scheme that works in conjunction with PTS which is called the Preemption Threshold Scheduling with Dynamic Reclamation (*PTS-DR*) algorithm. 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. Algorithm 2 describes the PTS-DR scheme. 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)* [27]. 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. Though we use earlier proposed data structures, we cannot use the slack reclamation policy proposed in [27] under the PTS policy. 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 must have a priority no smaller than the priority of all jobs in the ready queue.

We use the following notation and definitions in the PTS-DR algorithm as explained below. The notation is similar to the work in [27, 40].

- $R_i^r(t)$: 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 followed by the 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 priority and the priority of the run-time being the same, we show that all tasks meet the deadlines. The proof has a similar flavor to the work (and proofs) on task synchronization [27, 40].

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

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

On arrival of a new task τ_i :
Let processor be executing τ_c;
$R_i^r(t) \leftarrow \frac{C_i}{\eta_i}$;
Add task τ_i to Ready Queue;
if ($\tau_c = NULL$) **then**
  return
end if
if ($P(\tau_i) > P(\tau_c)$ **and** $\pi(\tau_i) > \gamma(\tau_c)$) **then**
  Preempt task τ_c;
else
  if ($P(\tau_i) > P(\tau_c)$) **then**
    Inherit Priority(τ_c);
    setSpeed ($\frac{E_c^r(t}{R_c^r(t)+R_c^F(t)}$);
  end if
end if

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

On completion of task τ_i :
Restore priority (τ_i);
Add to FRT-list($R_i^r(t), P(\tau_i)$);

Proof. Suppose the claim is false. Let t be the first time that a run-time of a job or that (run-time) in the 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 (5). Thus each task instance completes no later than its deadline. □

5.3 Dynamic Slack Reclamation with Other Implementation Architectures

We also extend the dynamic slack reclamation algorithm to other implementation architectures that minimize the number of system wide execution threads. In addition to minimizing context switches, preemption threshold scheduling also enables partitioning the task sets into non-preemption sets. This has shown to reduce the memory requirement in real-time systems [4, 5]. Given static slowdown factors for the tasks, the algorithms described in the prior works can be used to partition the task sets. Dynamic scheduling in the previous section (Sect. 5) only focuses on minimizing context switches. We show that dynamic slowdown can be easily extended to other known implementation architectures based on preemption threshold scheduling. As shown earlier (Fig. 1), it is known that prior work on dynamic slack reclamation [2, 27] cannot be directly used in these architectures, and can result in tasks missing the deadline. We extend earlier work to incorporate dynamic slowdown in these architectures. We show that the property of

priority inheritance ensures all deadlines and can be easily incorporated as described next.

Gai et al. [5] propose using shared virtual resources managed by the stack resource protocol (SRP) [39] 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. Explicit 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 [4] discussed an implementation 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 threads is determined off-line, 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**. This is the additional rule required for dynamic slack reclamation, which comes from the work proposed in this paper.
- *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.

6 Experimental Setup

We implemented the proposed scheduling techniques in a discrete event simulator. To evaluate the effectiveness of preemptive threshold scheduling algorithms, we consider synthetically generated task-sets, each containing 10-20 tasks. Task periods were generated uniformly in the range [10 ms, 100 ms]. 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. Each task set was simulated for 200 seconds to compute the energy consumption of the system. Experiments were performed on various task sets and the average results are presented in the paper.

We use the power model for dynamic power consumption of CMOS circuits [41]. 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 [41], 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 [38, 13]. Similar to the Intel PXA processor family [42], where the average power consumption at maximum speed is 0.411 mW, we assume the processor power consumption to be 500 mW at maximum speed. We have normalized the operating speed and support discrete slowdown factors in steps of 0.1 in the normalized range.

To incorporate the context switch energy overhead, we consider the additional memory accesses in saving/restoring the task context as well as due to the additional cache misses resulting from a context switch. Typical embedded processors have cache sizes between 32 KB and 128 KB and we assume that the system has 32 KB of separate instruction and data cache. On the context switch, the instruction and data required by a program are usually not present in the cache and we assume that 8 Kb of information is transferred to both the instruction and data cache. An energy cost of 15nJ [11] per memory access results in a total of 2 * 8K * 15 nJ = 0.24 mJ. The data cache can result in additional energy overhead, since dirty data needs to be flushed to memory before storing new data. We note that there are several other sources of overhead such as saving and restoring the task context. The overhead can vary with each context switch and we assume the average energy overhead per context switch to be 0.2 mJ. When we vary the task execution time, to reduce processor utilization (at full speed), there is relatively less load and less instructions are executed. To have a constant cache miss ratio when

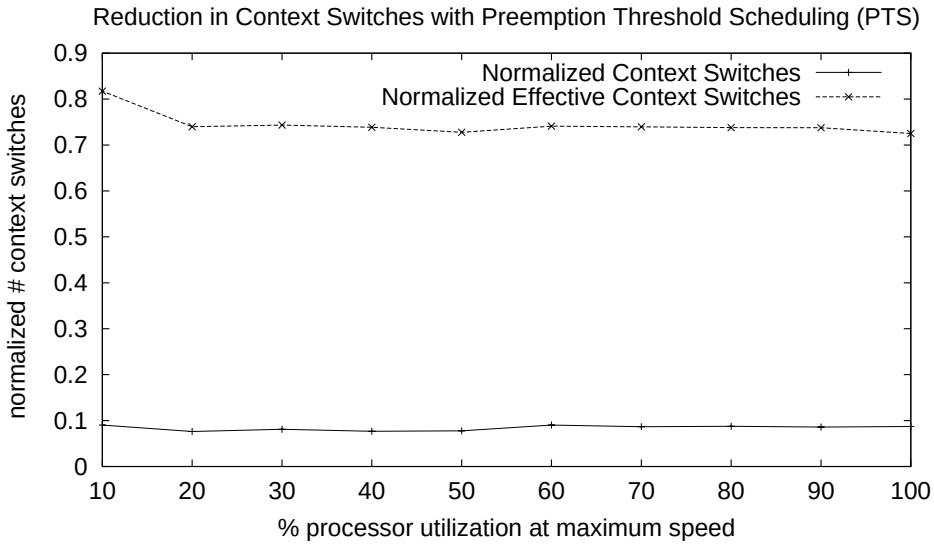

Fig. 2. Number of context switches under preemption threshold scheduling normalized to preemptive scheduling.

the amount of computation reduces, we proportionately scale (lower) the cache miss penalty when task execution times are lowered. Fewer computations imply slowdown and we proportionately reduce the cache miss penalty with slowdown. This assumption results in a conservation bound on the cache miss penalty.

6.1 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. Note that the slowdown factors can be computed with any known slowdown algorithm. Without loss of generality, 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 both,

- Preemptive Scheduling (PS) and
- 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 2 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 utilization at maximum speed along the X-axis and the normalized context

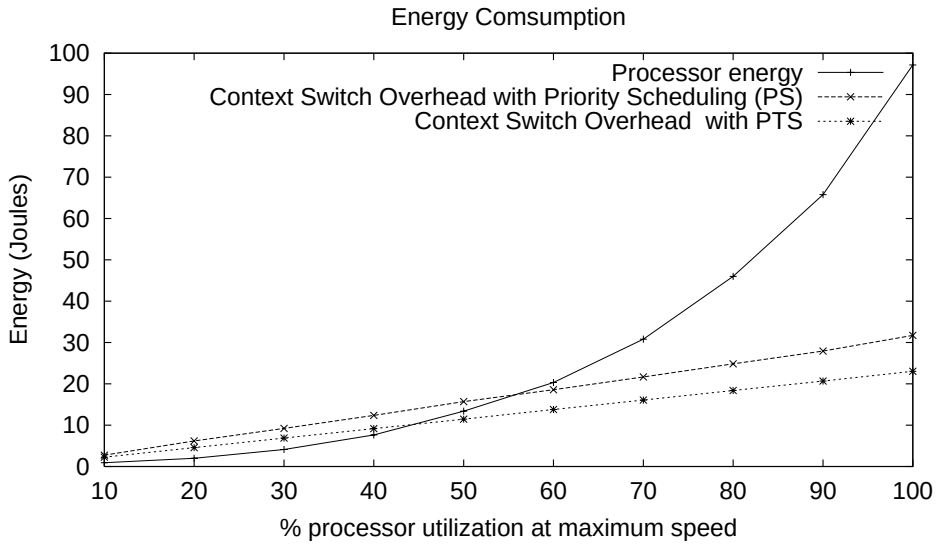

Fig. 3. The processor energy consumption and the energy overhead due to effective context switching. PTS reduces the context switching overhead which results in energy savings compared to priority scheduling.

switches along the Y-axis. The execution time for each task is its WCET at maximum speed. We see that the context switch reduction is independent of the task utilization at maximum speed. Processor slowdown increases the utilization to close to 100% and thus the context switches remain the same. We see that PTS considerably decreases the context switches, with an average 90% reduction in the context switches. We show that the context switches are reduced even in the presence of slowdown, which has not been shown in prior works.

A change in the locality of references is the main cause for increased cache miss ratio. The locality changes not only on a context switch but also when a task resumes execution on completion of another task. A change in the execution 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 lower than a context switch, a percentage decrease in the effective context switching gives a good measure of the gains achieved by preemption threshold scheduling. Figure 2 shows the normalized context switches as well as the normalized effective context switching between the two scheduling techniques. The two metrics have a similar pattern, however note that the context switches reduce by up to 90% whereas the effective context switch reduction is by 25% only. 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. The contribution of the processor and memory access energy is shown in Fig. 3. At lower utilization, the amount of work (number of instructions) decrease and the cache

miss penalty is decreased proportionately, as discussed earlier. Thus we see a linear reduction in the cache miss penalty with decreased utilization (Fig. 3). Lower utilization enables processor slowdown through dynamic voltage scaling, which reduces the processor energy consumption. PTS reduces the context switch overhead, which results on an average 5% - 10% energy savings due to reduced memory accesses. 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. Preemptive scheduling also has an additional overhead due to the intermittent (and additional) memory accesses, which reduce the chances of switching the memory to a low power mode. Thus we see that preemptive threshold scheduling is increasingly important for energy efficient execution of real-time systems.

6.2 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, μ = (WCET+BCET)/2 and a standard deviation, σ = (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 processor utilization (U) at maximum speed. Figure 4 shows the energy gains for different values of BCET at U=80% and U=40%. In the figure, we compare the energy consumption of both :

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

The energy consumption includes both the processor energy along with the energy overhead due to context switching. A steady decrease in the energy consumption is seen with a decrease in BCET. As the task execution time is decreased, there is more slack which results in decreasing the energy consumption by operating at a lower voltage. Note that under the PTS-DR scheme, the processor can consume more energy while minimizing the context switches. This is due to the fact that PS-DR can reclaim all the higher priority slack at all times. On the other hand, under PTS-DR when a higher priority task (but not a higher preemption level task) does not preempt the current task execution, the current task inherits the higher priority until it completes. This higher priority reduces the slack that can be reclaimed, thereby potentially increasing the processor energy consumption. 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. This is explicitly seen at BCET of 10% in Fig. 4(a). However, the energy savings achieved by minimizing the context switches result in reducing the overall energy consumption under PTS-DR. We see that PTS-DR leads on an average 10% energy savings over PS-DR.

Furthermore, the time delay incurred due to excess cache misses in PS-DR will reduce the dynamic slack generated in PS-DR, thereby increasing the energy gains of PTS-DR. It has been shown that the processor utilization drops as much as 8% with

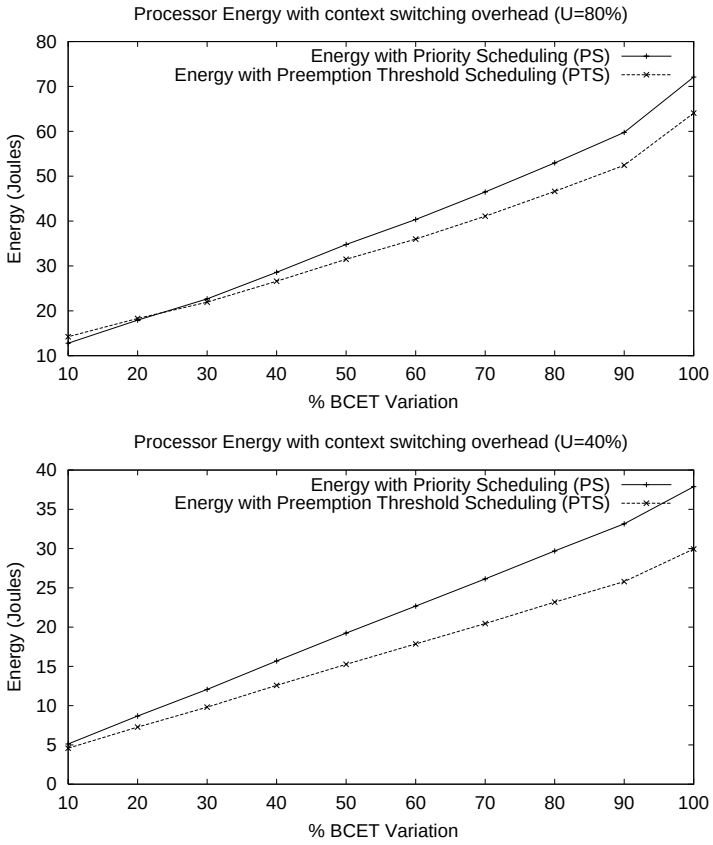

Fig. 4. Total Energy Consumption with Dynamic Slack Reclamation, including the context switching overhead. Comparison on PTS-DR with PS-DR for task sets with (a) Utilization at maximum speed, $U = 80\%$ and (b) Utilization at maximum speed, $U = 40\%$.

excessive context switching [43]. With the increasing time and energy overhead associated with an effective context switch, the total energy gains will keep increasing. This will further enhance the gains achieved by PTS-DR by reducing the context switches.

7 Summary and Future Work

We have presented algorithms to integrate task slowdown and preemption threshold scheduling. Preemption threshold scheduling is important in reducing the context switching among tasks as well as lowering the 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 that even in the presence of slowdown, PTS leads to an average 90% reduction of the context switching overhead compared to preemptive scheduling. We also present a dynamic slack reclamation algorithm that works in conjunction with preemption threshold scheduling. Dynamic slack reclamation results on an average 10% energy savings compared to slack reclamation with preemptive scheduling. These techniques are energy efficient and easy to implement in real-life systems. These scheduling techniques will increase the energy efficiency of systems and will have a great impact on the energy utilization of portable devices.

Acknowledgments

The authors acknowledge support from National Science Foundation (Award CCR-0098335) and from Semiconductor Research Corporation (Contract 2001-HJ-899). We would like to thank the reviewers for their useful comments.

References

1. Aydin, H., Melhem, R., Mossé, D., Alvarez, P.M.: Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In: Proceedings of EuroMicro Conference on Real-Time Systems. (2001)
2. Aydin, H., Melhem, R., Mossé, D., Alvarez, P.M.: Dynamic and aggressive scheduling techniques for power-aware real-time systems. In: Proceedings of IEEE Real-Time Systems Symposium. (2001)
3. Wang, Y., Saksena, M.: Scheduling fixed priority tasks with preemption threshold. In: Proceedings of IEEE International Conference on Real-Time Computing Systems and Applications. (Dec 1999)
4. Wang, Y., Saksena, M.: Scalable multi-tasking using preemption threshold scheduling. In: Proceedings of IEEE Real-Time Technology and Applications Symposium. (Jun 2000)
5. Gai, P., Lipari, G., di Natale, M.: 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. (2001)
6. Bradford, E.G.: (Runtime: Context switching) http://www-106.ibm.com/developerworks/linux/library/l-rt9.
7. Manegold, S., Boncz, P.: (Cache-memory and tlb calibration tool) http://homepages.cwi.nl/ manegold/Calibrator/DB/.
8. Mogul, J.C., Borg, A.: The effect of context switches on cache performance. In: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, ACM Press (1991) 75–84
9. Agarwal, A., Hennessy, J., Horowitz, M.: Cache performance of operating system and multiprogramming workloads. ACM Transactions on Computer Systems **6** (1988) 393–431
10. Fromm, R., Perissakis, S., Cardwell, N., Kozyrakis, C.E., McGaughy, B., Patterson, D.A., Anderson, T.E., Yelick, K.A.: The energy efficiency of IRAM architectures. In: Proceedings of International Symposium on Computer Architecture. (1997) 327–337
11. Lee, H.G., Chang, N.: Energy-aware memory allocation in heterogeneous non-volatile memory systems. In: Proceedings of International Symposium on Low Power Electronics and Design. (Aug. 2003) 420–423

12. Zucker, D., Lee, R., Flynn, M.: Hardware and software cache prefetching techniques for MPEG benchmarks. IEEE Trans. on Circuits and Systems for Video Technology **10** (2000)
13. Intel XScale Processor: (Intel Inc) *(http://developer.intel.com/design/intelxscale)*.
14. Zhang, C., Vahid, F., Najjar, W.: A highly configurable cache architecture for embedded systems. In: Proceedings of International Symposium on Computer Architecture. (2003) 136–146
15. Brock, B., Rajamani, K.: Dynamic power management for embedded systems. In: Proc. of the IEEE Int'l SOC Conference, Portland, Oregon. (Sept. 2003)
16. Kumar, R., Farkas, K., Jouppi, N., Ranganathan, P., Tullsen, D.: Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In: Proceedings of International Symposium on MicroArchitecture. (Dec. 2003)
17. Yao, F., Demers, A.J., Shenker, S.: A scheduling model for reduced CPU energy. In: Proceedings of IEEE Symposium on Foundations of Computer Science. (1995) 374–382
18. Quan, G., Hu, X.: Minimum energy fixed-priority scheduling for variable voltage processors. In: Proceedings of Design Automation and Test in Europe. (2002)
19. Yun, H., Kim, J.: On energy-optimal voltage scheduling for fixed-priority hard real-time systems. Trans. on Embedded Computing Sys. **2** (2003) 393–430
20. Kwon, W., Kim, T.: Optimal voltage allocation techniques for dynamically variable voltage processors. In: Proceedings of the Design Automation Conference. (2003) 125–130
21. Shin, Y., Choi, K., Sakurai, T.: Power optimization of real-time embedded systems on variable speed processors. In: Proceedings of International Conference on Computer Aided Design. (Nov. 2000) 365–368
22. Gruian, F.: Hard real-time scheduling for low-energy using stochastic data and dvs processors. In: Proceedings of International Symposium on Low Power Electronics and Design. (Aug. 2001) 46–51
23. Jejurikar, R., Gupta, R.: Optimized slowdown in real-time task systems. In: Proceedings of EuroMicro Conference on Real-Time Systems. (Jun. 2004)
24. Yan, L., Luo, J., Jha, N.K.: Combined dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems. In: Proceedings of International Conference on Computer Aided Design. (Nov. 2003)
25. Jejurikar, R., Pereira, C., Gupta, R.: Leakage aware dynamic voltage scaling for real-time embedded systems. In: Proceedings of the Design Automation Conference. (Jun. 2004)
26. Pillai, P., Shin, K.G.: Real-time dynamic voltage scaling for low-power embedded operating systems. In: Proceedings of 18th Symposium on Operating Systems Principles. (2001)
27. Zhang, F., Chanson, S.T.: Processor voltage scheduling for real-time tasks with non-preemptible sections. In: Proceedings of IEEE Real-Time Systems Symposium. (Dec. 2002)
28. Kim, W., Kim, J., Min, S.L.: 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. (Mar. 2002)
29. Rusu, C., Melhem, R., Mosse, D.: Maximizing rewards for real-time applications with energy constraints. In: ACM Transactions on Embedded Computer Systems. (accepted)
30. Luo, J., Jha, N.: Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In: Proceedings of International Conference on Computer Aided Design. (Nov. 2000) 357–364
31. Gruian, F., Kuchcinski, K.: LEneS: task scheduling for low-energy systems using variable supply voltage processors. In: Proceedings of the Asia South Pacific Design Automation Conference. (Jan. 2001)
32. Zhang, Y., Hu, X.S., Chen, D.Z.: Task scheduling and voltage selection for energy minimization. In: Proceedings of the Design Automation Conference. (2002)

33. Liu, J., Chou, P.H., Bagherzadeh, N.: Communication speed selection for embedded systems with networked voltage-scalable processors. In: Proceedings pf International Symposium on Hardware/Software Codesign. (Nov. 2002)
34. Liu, J., Chou, P.H.: Energy optimization of distributed embedded processors by combined data compression and functional partitioning. In: Proceedings of International Conference on Computer Aided Design. (Nov. 2003)
35. Kim, S., Hong, S., Kim, T.: Integrating real-time synchronization schemes into preemption threshold scheduling. In: Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. (2002) 145–152
36. Liu, J.W.S.: Real-Time Systems. Prentice-Hall (2000)
37. Transmeta Crusoe Processor: (Transmeta Inc) *(http://www.transmeta.com/technology)*.
38. IBM 405LP Processor: (IBM Inc) *(http://www-3.ibm.com/chips/products/powerpc/cores*.
39. Baker, T.P.: Stack-based scheduling of realtime processes. Journal of Real-Time Systems **3** (1991) 67–99
40. Jejurikar, R., Gupta, R.: Dual mode algorithm for energy aware fixed priority scheduling with task synchronization. In: Workshop on Compilers and Operating System for Low Power. (Sept. 2003)
41. Weste, N., Eshraghian, K.: Principles of CMOS VLSI Design. Addison Wesley (1993)
42. Intel PXA250/PXA210 Processor: (Intel Inc) *(http://www.intel.com)*.
43. Gopalakrishnan, R., Parulkar, G.M.: Bringing real-time scheduling theory and practice closer for multimedia computing. In: ACM SIGMETRICS Conference. (May 1996) 1–12