Preemption Handling and Scalability of Feedback DVS-EDF

Yifan Zhu and Frank Mueller *
Department of Computer Science/Center for Embedded Systems Research
North Carolina State University,Raleigh, NC 27695-7534
mueller@cs.ncsu.edu, phone: +1.919.515.7889, fax: +1.919.515.7925

ABSTRACT

Power-aware scheduling methods are a promising method to exploit the voltage and frequency scaling features of modern processors. We have devised a novel dynamic voltage scaling (DVS) scheme under earliest-deadline first (EDF) preemptive scheduling for hard real-time systems. This paper goes beyond our previous results [5] and makes the following contributions. First, we present the details of slack scheduling and address the challenges of preemption handling in DVS scheduling. Second, we present new results of our DVS scheme that demonstrate the scalability of our approach for task sets of different sizes. The results demonstrate that our DVS scheduling scheme provides up to 37% additional savings of the best published prior work with low runtime complexity and it scales for varying number of tasks.

Keywords

Real-Time Systems, Scheduling, Dynamic Voltage Scaling

1. INTRODUCTION

Energy consumption is a major concern for mobile embedded systems due to their limited battery lifetime. In embedded hard real-time systems, temporal constraints due to hard deadlines have to be met as well. The objective of this work is to exploit energy conservation due to DVS / dynamic frequency scaling (DFS) while guaranteeing feasible schedules. Since the energy consumption scales linearly and quadratically with frequency and voltage modulations, respectively, the combination of DFS and DVS can result in significantly lower power consumption. Real-time schedulability theory [15, 1, 21, 2, 3, 23] generally relies on *a prior* knowledge about the worst-case execution time (WCET) of each task, either statically or at admission time. However, experiments show a wide variation between longest and shortest execution times for many embedded applications, ranging between 30% and 89% of the WCET [24, 26].

Prior work has shown the potential to save energy by combining these scaling techniques with operating system scheduling, and significant savings have been reported for general-purpose computing systems [6, 9, 13, 16, 18, 25, 20, 8] as well as real-time systems [11, 12, 14, 22, 19, 10, 4, 17, 7] detailed in the related work section. Our work goes beyond the techniques explored for real-time systems in these previous studies. We contribute a novel approach for exploiting the slack time of a schedule and handle preemption through reservation schemes. Slack time is generated by actual executions of tasks that complete under budget with respect to their WCET. Our experiments demonstrate that the resulting energy savings exceed those of previously published work, they more closely resemble the optimal case, and they scale for varying number of tasks.

Our work is based on the widely used variation of EDF scheduling under frequency scaling. Consider a set of n tasks T_i with periods P_i and WCETs C_i. Let $\alpha = \frac{f_i}{f_m}$ denote the scaling factor representing the fraction of the current processor frequency f_i over the maximum frequency f_m:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq \alpha \tag{1}$$

Our DVS methods takes a novel approach to frequency scaling for EDF. Instead of assuming uniform scaling over all *future* tasks as in previous work, we scale the *current* task T_k (incidentally also the task with the earliest deadline). The remaining tasks are assumed to execute at maximum frequency f_m, *i.e.*, $\alpha = 1$. This can be expressed as:

$$\alpha^{-1} \frac{C_k}{P_k} + \sum_{i \in \{1,\ldots,n\} \setminus \{k\}} \frac{C_i}{P_i} \leq 1 \tag{2}$$

The motivation for only scaling the current task is that average execution times are typically smaller than worst-case execution times C_i, as explained before. If the current task finishes early, its slack can be used by the next task to scale frequencies again, and so on. Hence, early scaling is likely to leave enough potential for later scaling. In addition, slack due to future idle time can be utilized for scaling as well, as detailed next.

2. IDLE TIME UTILIZATION

In our DVS scheme, there are two opportunities for slack generation. One is due to early completion of tasks, which is calculated and converted to slack dynamically. The other is due to idle time resulting from idle slots in under-utilized system configurations and is determined statically. In our EDF-based scheduling, a maximum of 100% system utilization can be achieved in theory. But in practice, even the worst-case utilization of realistic real-time systems is generally lower than 100%. Here, we take a novel approach to convert the under-utilized system time slots into slack. A new task T_{n+1}, called idle task, is introduced to the task set to fill the gap between the actual utilization and 100% utilization. It is called the idle task since its actual execution time is always zero while its worse-case execution time is not. In other words,

$$P_{n+1} = P_1, C_{n+1} = P_{n+1}(1-U), c_{n+1} = 0. \tag{3}$$

Notice that any other choice of idle task periods is legal. Most notably, the shortest period of any task P_1 and the longest one P_n are interesting choices. We consider these options since they affect the complexity of our scheduling scheme, as detailed later. The idle task always completes

*This work was supported in part by NSF grant CCR-0208581.

early, thereby providing slack for other tasks in the task set. Most significantly, we know in advance the number of idle time slots (calculated in constant time). The total slack S generated by idle task T_{n+1} for the interval $[t1..t2]$ is:

$$idle(t1...t2) = \sum_{t1..t2} idle\ slots \qquad (4)$$

This idle time utilization approach is intriguing, not just because of its simplicity, but also because it naturally fits our slack-passing scheme. We choose the shortest task period in the task set as the idle task's period to ensure that there is at least one idle time slot lying between any actual task's invocation to provide slack for that task.

3. SLACK GENERATION

Slack passing and new slack generation are key techniques in our scheduling scheme. For n tasks, the algorithm takes $O(1)$ or $O(n)$ time for an idle task period equal to the shortest and longest task period, respectively. Slack passing is based on the observation that slack generated by one task will usually not be exhausted during the task's execution even if the task has been scaled as much as possible. The remaining slack (or part of the remaining slack) can be passed on to the next task if some conditions are satisfied. This slack is further adjusted and utilized to scale the next task to suit the actual frequency levels, as depicted in Figure 1.

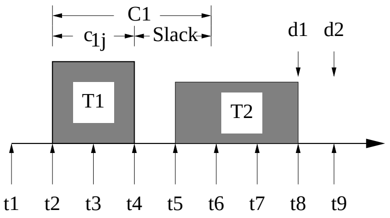


Figure 1: Slack Generation

Let task T_1 with WCET C_1 and deadline t8 execute its j^{th} invocation with an actual execution time of c_{1j}. Assume that when T_1 was invoked at time t2, it inherited a total slack of S from its previous tasks. T_1 was then scaled to a suitable level with that slack and completed at time t4. The difference between C_1 and c_{1j} is the new slack dynamically generated by T_1. So the total slack available at t5 is

$$S = S + C_1 - c_{1j} \qquad (early\ completion) \qquad (5)$$

Note the fact that the actual execution time c_{1j} may be less than, equal to, or greater than the worst-case execution time C_1 because of task scaling. If $C_1 > c_{1j}$, Equation 5 just adds the slack produced by the early completion of T_1 into the total slack. When $C_1 < c_{1j}$, Equation 5 will in fact reduce the total slack because the task exceeded its slot alloted at the highest frequency under EDF. (Notice that exceeding a task's WCET is feasible under DVS-EDF due to slack exploitation as long as the available slack is not exceeded as well.) The adjusted total slack (Equation 5) cannot be passed in full to the next task T_2. Due to EDF scheduling, the next task T_2 has a deadline equal to or larger than T_1's deadline t8. Hence, the amount of slack available to T_2 depends on T_2's release time and deadline. This is based on the observation that slack beyond a task's release time and deadline cannot be used by this task. Let r_2 be T_2's release time and t9 be its deadline. First, T_2 can utilize all of the slack ($r_2 \leq t5$), part of the slack ($t5 \leq r_2 \leq t8$), or none of the slack ($r_2 \geq t8$). More abstractly, we can express these portions in terms of the release time r_{cj} within the dynamic EDF schedule of the current task instance as well as the initiation time I_{pk} and completion time F_{pk} within the static (worst-case EDF) schedule of the predecessor task's instance.

$$s_{cj} = \begin{cases} C_p - c_{pk} & if\ r_{cj} \leq I_{pk} + c_{pk} \\ F_{pk} - r_{cj} & if\ I_{pk} + c_{pk} < r_{cj} < F_{pk} \\ 0 & if\ r_{cj} \geq F_{pk} \end{cases} \qquad (6)$$

Based on the above adjustment, T_2 can further utilize the slack produced by the idle task before T_2's deadline t9 but after T_1's deadline t8. We use formula 4 to find this slack portion and add it to the total slack (with constant overhead):

$$S = S + idle(d1..d2) \qquad (idle\ slots) \qquad (7)$$

The idle function simply aggregates idle slots of the static schedule in the specified interval while the old slack is derived from Equation 5. In summary, Equation 7 gives us the total slack that can be utilized by T_2.

4. REFINED PREEMPTION HANDLING

When preemption occurs, the preempted task will relinquish its remaining slack and pass it on to the next task, just as during task completion. It again follows a greedy scheme in that we try to pass as much slack as possible to scale running tasks and speculate on early completion to aggregate more slack. But there are two differences here. First, the preempted task itself cannot generate any slack based on its own execution at preemption points since the completion time is unknown (in the future). Hence, no additional slack is added to its inherited total slack. Second, the preempted task still needs some time to complete its execution in the future. The remaining execution time must be reserved in advance to avoid future deadline misses caused by over-exploiting slack by other tasks. At preemption, the expected remaining execution time of the preempted task $left_{ij}$ is:

$$left_{ij} = C_i - c_{ij} \times \alpha\prime \qquad (8)$$

Our slack passing scheme promises that the preempted task will not miss its deadline by reserving corresponding slack:

$$S = S - left_{ij} \qquad (future\ slots) \qquad (9)$$

The old slack is derived from Equation 7 and the resulting slack S can be passed to the next task.

Future slot allocation in this manner is essential to ensure the feasibility of the scheduling under DVS. Future slots will be allocated only if the static schedule does not include sufficient slots for the preempted task's instance between the preemption point and its deadline. We devised multiple schemes for reserving these slots.

- Forward sweep: When a task $T1$ is preempted and requires $left_{1j}$ future slots, the preempting task $T2$ deduces this amount from its available slack S. If $left_{1j} > S$, then $T2$ remains without slack. If another task $T3$ is initiated, the calculation repeats itself.
- Backward sweep: Future slots of $T1$ are allocated in idle slots within the static schedule from its deadline $d1$

backwards. Any of these idle slots become unavailable for slack generation, *i.e.*, these slots are excluded in Equation 7.

An example is depicted in Figure 2. The upper time line of idle slots presents a excerpt of the static schedule that depicts idle task allocations, only. The lower time line shows the dynamic schedule of tasks. Upon release of $T2$ at t2, $T1$ is preempted. Let us assume that $T1$ does not have sufficient static slots (three slots) beyond t2 to finish its execution. Hence, it has to rely on future idle slots. During $T2$'s execution, $T3$ is released. Both $T2$ and $T3$ have smaller deadlines than $T1$ ($d2 < d3 < d1$). Subsequently, $T1$ only resumes some time after $T3$ completes.

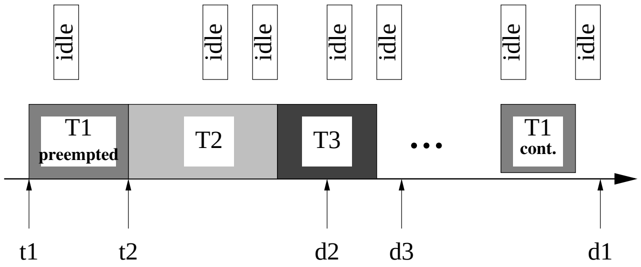


Figure 2: Future Slot Reservation

Future slot allocation of $T1$ then depends on the chosen scheme. The forward sweep results in zero idle slack for $T2$ and $T3$ since idle slots during the tasks' periods are not sufficient to cover $T1$'s future needs of three slots at the respective invocation times. The backward sweep, on the other hand, reserves the last 3 idle slots (from d1 backwards), such that $T2$ and $T3$ may consume at least two and one idle slots for scaling, respectively, even if they use up their time quantum in full.

Overall, the forward sweep is not as greedy as the backward sweep in the sense that earlier tasks may not be scaled due to $T1$'s future slots. A forward sweep is likely to result in zero slack for the preempting task $T2$ if $P2 << P1$, *i.e.*, if its period is much shorter. There are simply fewer idle slots available, which may not suffice to cover $T1$'s future requirements. More idle slots past $d2$ will be required in this case. The backward sweep always results in the most greedy approach in delaying the needs of $T1$ as long as possible. This is consistent with the observation that early completion is likely to generated slack for each task, which is inherent to our base algorithm.

5. FEEDBACK SCHEME

In a previous paper [5] we introduced a greedy task partitioning scheme. In the ideal case, an energy optimal schedule can be achieved when the processor changes its frequency level frequently, even during the execution of a task. However, when taking into consideration the frequency and voltage scaling overhead, frequent changes can inflict considerable overhead, thereby resulting in anything but the optimal schedule. To address this dilemma, we restrict the number of frequency changes for each task to be at most two. Any task is split into two parts (C_A and C_B). While the second part (C_B) always executes at the maximum frequency level, the first part (C_A) can execute at the lowest feasible frequency level, which depends on the ratio of C_A and C_B. Initially, C_A is chosen as 50% of the WCET. Half of the task's execution is budgeted at a low frequency, half of it is reserved at the maximum frequency. The task can still meet its deadline, even if the worst case is exhibited. Initially, the energy savings may already be significant but are likely to digress from the optimal case due to inappropriate estimation of the actual execution time. Over time, we replace C_A with the actual execution time of the task through execution time fed back after each task completion. The mean of execution times over past executions is utilized to anticipate future C_A portions. On the average, this scheme allows us to complete the entire task's budget at a low frequency level, which closely approximates the optimal energy-saving schedule. Let C_{Ai} be the anticipated C_A value when instance i of a task is released. We define the following equations to get the anticipated C_A value for instance $i+1$:

$$\begin{aligned} C_{A1} &= 0.5 \times WCET \\ C_{A(i+1)} &= \frac{C_{Ai} \times (i-1) + c_i}{i}, i \geq 1 \end{aligned} \tag{10}$$

Here, c_i is the actual execution time of the task's instance i. Each time an instance completes execution, its actual execution time is fed back and aggregated to anticipate the next instance's actual execution time, which is further used to calculate an ideal scaling factor for that task.

6. PREEMPTION EXAMPLE

An algorithmic description of the optimistic scheduling technique derived in the last section is depicted in Figure 3. The algorithm is a refinement of our previous work [5] and reflects the backward sweep scheme. A forward sweep changes the calculation of the slack during task activation for the preemption case, as explained before. We use the following notation:

- T_{ij}: the j-th instance of task T_i
- ij, pk : indices for the current and previous tasks relative to T_{ij}
- now: the current time
- r_{ij}: the release time of T_{ij}
- d_{ij}: the deadline of T_{ij}
- C_i: the WCET of T_i (without scaling)
- c_{ij}: the actual execution time of T_{ij} up to now (with scaling)
- $left_{ij}$: the remaining WCET of T_{ij} (without scaling)
- $slack$: system current slack
- $idle(t1..t2)$: the amount of idle slots between time [t1,t2]
- $slots(T_{ij}, t1..t2)$: the amount of time slots reserved for T_{ij} in the worst case between time [t1,t2]

The following example illustrates the behavior of our feedback-DVS algorithm. Consider the task set and actual execution times in Figures 4(a) and 4(b), respectively. This task set results in the traditional EDF schedule without voltage scaling depicted in Figure 5(a). Figure 5(b) shows a snapshots for preemption under our feedback DVS scheme. At time 240, $T1$ preempts $T3$ and inherits slack (2.0ms) from $T3$. Since $T3$'s remaining execution time is 0.5ms, $T1$ reserves the corresponding time unit from its inherited slack, which results in a final available slack of 1.5ms and makes $T1$ run at 50% frequency. At time 242, $T1$ completes and generates one unit of slack. The total slack becomes 1.5+1=2.5ms

Procedure Initialization
  for each $T_k \in \{T_1, T_2, \ldots, T_n\}$ **do**
    $C_{avg\_k} \leftarrow C_k/2$
    $left_{k0} = C_k$
  $U \leftarrow \frac{C_1}{P_1} + \frac{C_2}{P_2} + \ldots + \frac{C_n}{P_n}$
  $P_{n+1} \leftarrow P_1, C_{n+1} \leftarrow P_1 \times (1-U), c_{n+1} \leftarrow 0\}$
  let $slack \leftarrow 0$

Procedure TaskActivation($\mathbf{T_{ij}}$)
  if processor was idle for d **then**
    $slack \leftarrow slack - d$
  if T_{pk} was preempted/interrupted **then**
    $slack \leftarrow slack - idle(d_{ij}..d_{pk})$
    if $left_{pk} > slots(T_{pk}, now..d_{pk})$**then**
      $reserve_{pk} \leftarrow left_{pk} - slots(T_{pk}, now..d_{pk})$
      $slack \leftarrow slack - reserve_{pk}$
  else (T_{pk} *completed execution*)
    if $now > d_{pk}$ **then**
      $slack \leftarrow slack - idle(d_{pk}, now)$
    $slack \leftarrow slack + idle(d_{pk}..dij)$
  $\alpha\prime \leftarrow \min\{\frac{f_1}{f_m}, \ldots, \frac{f_m}{f_m} | \frac{f_i}{f_m} \geq \frac{C_{avg\_i}}{C_{avg\_i}+slack}\}$
  if $(\alpha\prime = 1)$ **then**
    $C_A \leftarrow 0$
  else
    $C_A \leftarrow slack \times \alpha\prime/(1-\alpha\prime)$
  SetInterrupt($T_i, C_A/\alpha\prime$)
  SetFrequency($\alpha\prime$)

Procedure TaskCompletion($\mathbf{T_{ij}}$)
  if T_{ij} is preempted **then**
    $left_{i(j+1)} = C_i - c_{ij} \times \alpha\prime$
  else
    $slack \leftarrow slack - c_{ij} + C_i$
    $C_{avg\_i} \leftarrow (C_{avg\_i} \times (j-1) + c_{ij} \times \alpha\prime)/j$
    $left_{i(j+1)} = C_i$
    if$reserve_{ij} > 0$ **then**
      release $idle(now..d_{ij})$ up to $|reserve_{ij}|$

Figure 3: Pseudocode of Feedback DVS Scheme

and is passed on to the next task $T2$. $T2$ discovers one additional unit of idle slack between $T1$'s deadline 248 and $T2$'s own deadline 250 and adds it to the inherited slack. Using a total slack of 3.5ms, $T2$ runs at 25% frequency. Note that the 3.5ms slack still leaves a slack of 0.5ms reserved for the preempted task $T3$. At time 246, $T3$ resumes and runs at 25% frequency without missing its deadline. This demonstrates another property of our slack-passing scheme. Whenever any slack is reserved for the preempted task, the reservation can also be passed on to next tasks as long as the preempted task does not resume execution.

In the next section, we compare the results of our algorithm with another approach and demonstrate that our DVS scheme results in a schedule that matches the optimal energy consumption more closely independent of the number of tasks.

7. EXPERIMENTS

We implemented our algorithm in a simulation environment that supports EDF-based scheduling. We chose to implement idle periods equal to the shortest period and forward sweeping for future slot allocation upon preemption. In the same environment, we also implemented a look-ahead DVS algorithm [19], which is the best dynamic scheduling algorithm for energy conservation that we know of. In order to make a comparison, energy values produced in ideal optimal scaling levels without consideration of deadline misses are also computed, which provide theoretical lower bounds. We provide different frequency settings and assume the processor will scale to the lowest level during idle time since it is not realistic to put a processor into sleep mode for frequent task releases. We restrict ourselves here to report results based on four frequency settings and associated voltage levels, as depicted in Table 1. The choice of DVS levels is consistent with look-ahead DVS work [19] as well as experimental work with the StrongARM [20].

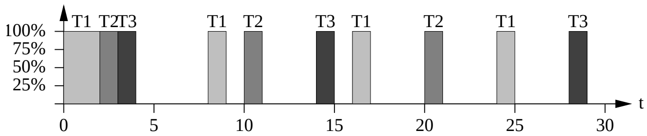
Task T_i	WCET C_i	Period P_i
1	3 ms	8 ms
2	3 ms	10 ms
3	1 ms	14 ms
idle	1 ms	4 ms

(a) Task Set

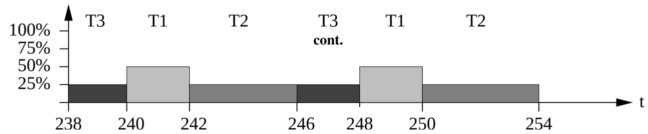
	Invocations	
Task T_i	c_{i1}	$c_{ij}, j > 1$
1	2 ms	1 ms
2	1 ms	1 ms
3	1 ms	1 ms
idle	0 ms	0 ms

(b) Actual Execution Times

Figure 4: Sample Task Set



(a) No Scaling



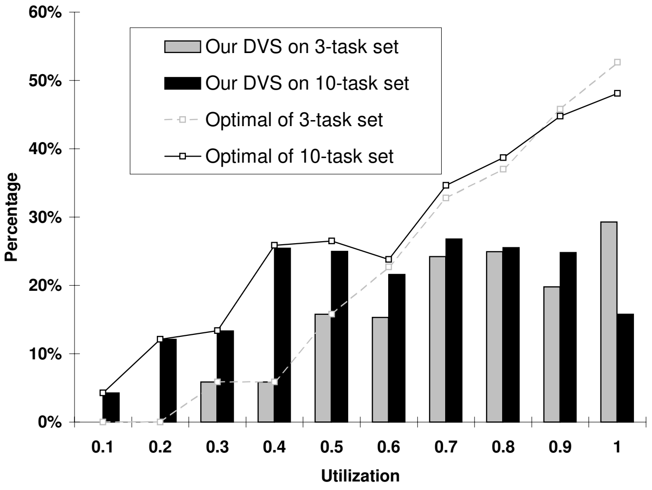
(b) Our Feedback DVS at Time 240

Figure 5: Sample Execution

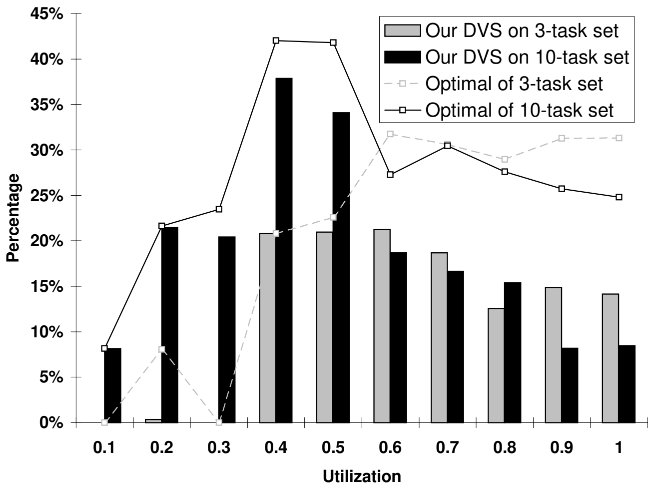
frequency	voltage
25%	2 V
50%	3 V
75%	4 V
100%	5 V

Table 1: Processor Model for Scaling

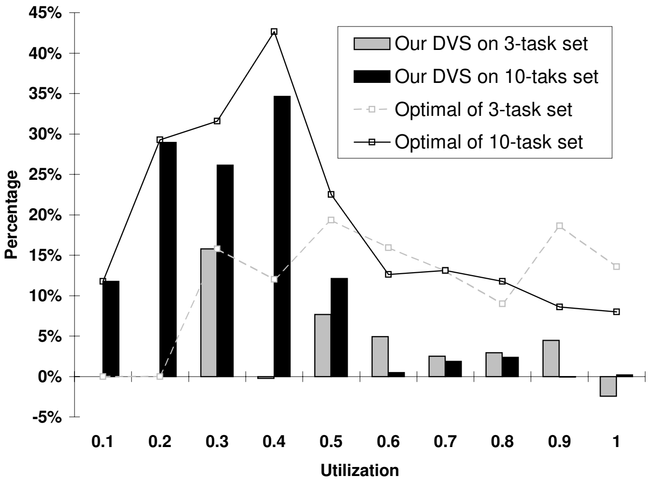
Due to the feedback of our DVS approach, energy results exhibited during the first hyperperiod are slightly worse due to the fact that its initial tasks cannot accurately predict actual execution times. After one hyperperiod, the values



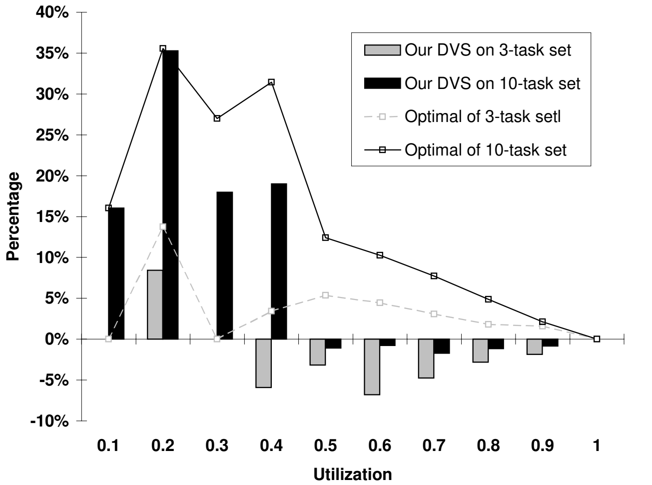
(a) Energy Consumption for 25% of WCET



(b) Energy Consumption for 50% of WCET



(c) Energy Consumption for 75% of WCET



(d) Energy Consumption for 100% of WCET

Figure 6: Relative Energy Consumption Savings of Our Feedback DVS over Look-Ahead DVS

approach a stable point, and scaling is more accurate resulting in higher energy savings. In the graphs, we reported the results for up to ten hyperperiods. Hence, our actual results for later hyperperiods are even slightly better than shown.

We experimented with different task sets consisting of 3 and 10 tasks. Task set utilizations were varied between 10% and 100% in increments on 10%. We also varied each task's actual execution time to be 25%, 50%, 75% and 100% of the WCET, respectively, to see the performance effects of our algorithm in different situations.

Figures 6(a-d) summarize the results of our comparison for 3 and 10 tasks. These figures depict the savings of (i) our DVS scheme and (ii) of the lower bound, in both cases *relative* to Pillai's approach [19]. In other words, Pillai's Look-Ahead scheme is equivalent to 0%. The optimal curve is an upper bound on the savings (or a lower bound on energy consumption) that can potentially be achieved by any DVS scheme.

For actual executions of 25% relative to the WCET, depicted in Figure 6(a), we observe close to optimal results of our DVS scheme up to 50% utilization with savings of up to 26% over the Look-Ahead method. The set of 10 tasks shows higher savings than the 3-task set. Higher utilizations also result in savings over Look-Ahead but here, the 3-task set is performing better.

The highest savings of 37% for our scheme can be observed for actual executions of 50% relative to the WCET and under 40% utlization, as shown in Figure 6(b). The trends for larger task sets to perform better under low utilization and vice versa can be observed again.

With 75% of the WCET (Figure 6(c)), most savings over Look-Ahead occur in low utilization cases while hardly any differences can be observed on the high utilization end. This trend continues for 100% of the WCET (Figure 6(d) where our Feedback DVS performs slightly (but not significantly) worse than the Look-Ahead method. These results are consistent when we consider the greedy nature of our approach. We speculate on early completion and greedily exploit all slack available at the earliest point in time. However, if tasks use 100% of their execution time, we cannot profit from early completion. Our scheme also exploits idle slots for scaling. But under high utilization (close to 100%), there is little to no idle time left for scaling. As mentioned before, actual executions are typically well below the worst case and utilizations typically stay well below 100%, *i.e.*, our feedback shines in the range that can typically be encountered in embedded real-time systems.

Overall, our Feedback-DVS outperforms the Look-Ahead DVS approach [19]. Both schemes reduce energy consumption considerably compared to the upper bound without scaling for tasks. Our scheme approximates the lower bound more closely than the Look-Ahead scheme). In summary, the experiments show that our approach scales for different sizes of task sets and exhibits considerable savings for most realistic scenarios of lower actual execution times up to medium utilizations.

8. CONCLUSION

We presented refined details of our Feedback DVS scheduling scheme. Our approach follows preemptive EDF scheduling for hard real-time systems and guarantees feasible schedules while preserving energy. This power-aware scheduling method exploits voltage and frequency scaling features of modern processors. We detailed the efforts of slack generation due to early completion, future idle slot budgeting and slot reservation at preemption points. We also presented results that demonstrate the strengths of our approach. Our Feedback DVS scheduling scheme provides up to 37% additional savings over the best published prior work, combined with low runtime complexity, and scales for varying number of tasks.

9. REFERENCES

[1] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *J. of Real-Time Systems*, 8:173–198, 1995.

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

[3] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer, 1997.

[4] J. Kim D. Shin and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. In *IEEE Design and Test of Computers*, March 2001.

[5] A. Dudani, F. Mueller, and Y. Zhu. Energy-conserving feedback edf scheduling for embedded systems with real-time constraints. In *ACM SIGPLAN Joint Conference Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02)*, page (accepted), June 2002.

[6] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power cpu. In *1st Int'l Conference on Mobile Computing and Networking*, Nov 1995.

[7] F. Gruian. Hard real-time scheduling for low energy using stochastic data and dvs processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'01*, Aug 2001.

[8] F. Gruian and Kuchcinski. Lenes: task scheduling for low-energy systems using variable voltage processors. In *Proceedings of ASP-DAC*, 2001.

[9] D. Grunwald, P. Levis, C. Morrey III, M. Neufeld, and K. Farkas. Policies for dynamic clock scheduling. In *Symp. on Operating Systems Design and Implementation*, Oct 2000.

[10] D. Mosse H. Aydin, R. Melhem and P.M. Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of 22nd Real-Time Systems Symposium*, December 2001.

[11] I. Hong, M. Potkonjak, and M. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processor. In *Int'l Conference on Computer-Aided Design*, Nov 1998.

[12] I. Hong, G. Qu, M. Potkonjak, and M. Srivastava. Synthesis techniques for low-power hard real-time systems on variable voltage processors. In *19th Real-Time Systems Symposium*, Dec 1998.

[13] C. Krishna and Y. Lee. Voltage clock scaling adaptive scheduling techniques for low power in hard real-time systems. In *6th Real-Time Technology and Applications Symposium*, May 2000.

[14] Y. Lee and C. Krishna. Voltage clock scaling for low energy consumption in real-time embedded systems. In *6th Int'l Conf. on Real-Time Computing Systems and Applications*, Dec 1999.

[15] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[16] J. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with pace. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, June 2001.

[17] D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low Power*, October 2000.

[18] T. Pering, T. Burd, and R. Brodersen. The simulation of dynamic voltage scaling algorithms. In *Symp. on Low Power Electronics*, 1995.

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

[20] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor, 2000.

[21] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, September 1990.

[22] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Int'l Conf. on Computer-Aided Design*, 2000.

[23] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling for Real-Time Systems*. Kluwer, 1998.

[24] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, November 2001.

[25] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *1st Symp. on Operating Systems Design and Implementation*, Nov 1994.

[26] W. Wolf. Smart cameras and embedded computing. seminar presentation, January 2002.