Improving Energy-Efficient Real-Time Scheduling by Exploiting Code Instrumentation

Thorsten Zitterell
Christoph Scholl

Institute of Computer Science
Albert-Ludwigs-University
Georges-Köhler-Allee 51
79110 Freiburg im Breisgau, Germany

Report 241, July 2008

Improving Energy-Efficient Real-Time Scheduling by Exploiting Code Instrumentation

Thorsten Zitterell Christoph Scholl
Albert-Ludwigs-University Freiburg, Germany
{tzittere, scholl}@informatik.uni-freiburg.de

Abstract

Dynamic Frequency and Voltage Scaling is a promising technique to save energy in real-time systems. In this work we present a novel light-weight energy-efficient EDF scheduler designed for processors with discrete frequencies which performs on-line intra- and inter-task frequency scaling at the same time. An intra-task scheduling scheme based on cycle counters of a processor allows the application of our approach to shared code of library functions and to task setups where only sparse intra-task information is available. Our 'Intra-Task Characteristics Aware EDF' (ItcaEDF) scheduler which aims to run with a low frequency by eliminating idle time and inter- and intra-task slack times was evaluated in an compiler, WCET analysis, and simulation framework. Our experiments show that state-of-the-art intra-task as well as inter-task frequency scaling approaches are clearly outperformed by our approach.

1 Introduction

Real-time systems like mobile multimedia devices have to meet high demands which are often competing. They require high performance under real-time constraints (for audio en-/decoding, e.g.), but also have to deal with limited resources like energy. Dynamic Voltage and Frequency Scaling (DVS/DFS) is an effective instrument to control the trade-off between system performance and energy-efficiency. Here, the operating system or the applications can decide whether the system should run at a higher frequency for higher performance but also with more power consumption – or at a lower frequency to save energy. When the timing behavior of real-time systems is specified by deadlines for individual tasks in the system, DFS (Dynamic Frequency Scaling) algorithms try to decrease the processor frequency under the constraint that deadlines are still guaranteed.

The concept of Dynamic Voltage and Frequency Scaling which changes the supply voltage and the processor frequency during run-time has been proposed in many publications, e.g., [2, 5, 9, 10, 16, 1, 11], as a technique to reduce energy consumption for systems with and without real-time constraints. Existing approaches to DVS can be divided into inter-task voltage scheduling and intra-task voltage scheduling.

Approaches to *inter-task* voltage scaling make use of the fact that computation times of tasks are usually not fixed due to different execution paths during run-time. Whereas standard scheduling approaches for real-time systems are based on fixed worst-case execution times for the tasks, inter-task voltage scaling makes use of 'inter-task slack times', which occur when a task instance terminates earlier than expected for the worst-case. For instance, Lee and Shin [7] adjusted the EDF scheduler [8] to the voltage scaling context. They presented an energy-efficient on-line EDF algorithm which requires only constant time for each context switch. In their approach voltage scaling is only performed when a task starts or resumes after another task which leaves some 'inter-task slack time' to the following task. [7] assumes a processor model which is able to provide a continuous spectrum of frequencies (limited by a maximum frequency f_{max}).

Whereas in many cases inter-task voltage scaling approaches succeed in reducing energy consumption, there are application scenarios where inter-task voltage scaling is less effective: Since voltage scaling can be applied only between two activations of different tasks, the granularity of inter-task voltage scaling may not be high enough, especially when the system consists of a small number of relatively long-running tasks (with one task as an extreme case).

In contrast to inter-task voltage scaling, *intra-task* voltage scaling algorithms perform voltage scaling during the execution of single tasks (and not between two activations of different tasks). Typically, there are many points in the program for a single task where frequency scaling is performed in case that execution time was saved (relative to the worst-case execution time). In that way, slack time for downsizing the processor frequency can be used much earlier than by inter-task voltage scaling and at a higher granularity). Compared to inter-task methods this may potentially lead to a more homogeneous distribution of used processor frequencies and thus to higher energy savings.[1] In [15] control flow analysis determines execution paths in a program whose traversal will save cycles and result in earlier task termination. Depending on the number of saved cycles the executable code is instrumented at *Voltage Scaling Edges (VSEs)*. Voltage scaling edges are restricted to branches in the control flow. At each edge the processor frequency is decreased by a given factor and the system can save energy. The basic concept has been further optimized by identifying earlier voltage scaling points using data flow information of the program [14] and by using profile information to estimate probabilities of branches [13].

The intra-task algorithms mentioned above concentrate on single-task environments and they consider processors with a continuous frequency range. Thus, these algorithms are able to enforce that task execution completes exactly at its deadline, provided that there is a frequency scaling point at each branch in the control flow. However, this is not a realistic scenario:

- Usually, processors do not offer a continuous frequency range, but a set of discrete

[1] Idle times, which indicate a non-optimal voltage scaling strategy, can not always be avoided by inter-task methods: In the extreme case, the slack produced by a task running much shorter than the worst case execution time, can not be used by other tasks, since no instances of other tasks have been released at finishing time of that task (even if processor utilization is maximal assuming worst-case timing).

frequencies. On processors with discrete frequencies the frequency required by the intra-task algorithm may not be available and the processor has to run at some higher speed, thus the task may finish *before* its deadline.

- Moreover, in real applications it does not make sense to place a frequency scaling point at *every* branch in the control flow because of the overhead both for frequency scaling itself and for computing saved cycles and scaling factors at run-time. For that reason an additional slack may occur.

- Finally, there may also be other reasons for the situation that a task is not able to keep account of every cycle it has saved compared to worst-case estimations during worst-case execution time (WCET) analysis: Especially for more complex architectures including caches and pipelines the execution times really needed may be smaller than the WCET assumptions, though tasks fail to detect these saved cycles at frequency scaling points.

Based on these observations the idea of our approach was to combine the advantages of inter-task and intra-task frequency scaling in an on-line real-time scheduler. More precisely, we propose a multi-level approach to DFS for a set of periodic tasks: The first level tries to eliminate idle times in case that the processor is not fully utilized. The optimal frequency for the task set is dynamically recalculated whenever tasks are created or removed. Whereas the first level is only based on worst-case execution times, the second and the third level take into account that the actual computation times are often much lower that the worst-case times: At level two we perform intra-task frequency scaling for each task individually. Finally, slack times produced by intra-task frequency scaling are used by inter-task frequency scaling at level three.

The idea of combining intra- and inter-task frequency scheduling has already been previously followed by Seo et al. [12], but in a completely different context: Seo et al. consider a fixed and finite set of a-periodic tasks (potentially including task dependencies) and they compute an optimal schedule using an off-line algorithm. Preemption is not allowed in their model, i.e., the schedule consists of starting times s_i and ending times e_i for each task τ_i. Intra-task scheduling is then based on branching probabilities and for a task τ_i it computes (under the assumption of a continuous frequency range) a frequency distribution which minimizes the average case energy consumption provided that the execution time will be exactly $e_i - s_i$.

In contrast, our approach handles *dynamic* sets of *periodic* tasks which are scheduled by an *on-line* algorithm. Moreover, our scheduler allows *preemption* and voltage scaling is performed for processors with a *discrete set of frequencies*. To the best of our knowledge our method is the first on-line algorithm combining inter- and intra-task voltage scaling.[2]

The contributions of our paper are as follows:

[2]It is not hard to see that our method can be easily changed into a preemptive on-line algorithm (assuming a discrete set of frequencies) for *a-periodic* tasks, too. For that purpose level 1 of our three-level approach and the on-line schedulability check have to be adjusted.

- We make intra-task frequency scaling applicable in a multi-task environment and we make use of it in a tight interaction with inter-task scheduling. We provide an on-line algorithm suitable for dynamically changing sets of periodic tasks and we support preemptive task execution.
- Thereby, we provide a simplified scheme for computing information necessary for rescaling in the intra-task context. This scheme is based on a cycle counter in the processor hardware and it makes it possible to directly instrument different execution paths within (nested) loops.
- Our scheme for code instrumentation is also suitable for shared code in libraries used by different tasks.
- We provide a method based on a realistic processor model with a set of discrete frequencies instead of a continuous range of frequencies. The method can directly be integrated into an operating system if overhead for context switches and for frequency scaling is considered according to Section 3.6.

It is important to note that our frequency scaling method achieves its goals adding only a constant overhead to each context switch and to each execution at a frequency scaling point.

The paper is organized as follows. We first give some preliminaries and definitions for inter- and intra-task real-time scheduling in Section 2. Section 3 presents the concepts and implementation of our three-level approach. Experimental results are described in Section 4. Finally, Section 5 concludes the paper.

2 Preliminaries

2.1 Processor and Hardware Model

The processor model used in this paper provides m different operating configurations $S = \{s_1, ..., s_m\}$ with $s_i = (f_i, v_i)$. Each pair (f_i, v_i) consists of a frequency f_i and a voltage v_i with $f_i < f_j$ for $1 \leq i < j \leq m$. Therefore, the minimum and maximum possible frequencies are f_1 and f_m, respectively. The hardware also provides a cycle counter $\check{C}_{CLK}$ which is automatically incremented during task execution and a frequency-independent clock providing the real time t.

2.2 Energy Model

In order to evaluate the energy-efficiency of our scheduler, we use the assumptions that the power consumption in CMOS circuits scales quadratically to the supply voltage ($P \propto f \cdot V^2$) [3]. Therefore, we compute the (dynamic) energy consumption E of the system by

$$E = const \cdot \sum_{s_i \in S} c_{f_i} \cdot v_i^2$$

provided that c_{f_i} cycles have been executed at frequency f_i or voltage v_i, respectively.

2.3 Earliest-Deadline-First Scheduler

An Earliest Deadline First (EDF) scheduler [8] always executes the task with the earliest deadline. A task τ_i is specified by its computation cycles $\check{C}_i$ and its period T_i. The symbol $\check{}$ will be used to denote variables containing cycle-based values. Thus, a task set is specified by $\Gamma = \{\tau_i(\check{C}_i, T_i), i = 1, .., n\}$. The phase ψ_i of a task is implicitly given by its activation. We assume that the relative deadline D_i of a task is the same as the period T_i, the absolute deadline d_i for a task instance increments with T_i for each instance.

In real-time systems, the worst-case execution cycles $\check{C}_i$ of a task τ_i for a specific processor *internally* depends on the task code and can be determined by static code analysis, for example. However, the period T_i of a task is *externally* specified by the system designer for a task function. Moreover, the worst-case computation time C_i of a task τ_i can be *derived* from the computation cycles $\check{C}_i$ if the execution frequency f of the processor is known: $C_i = \check{C}_i \cdot f^{-1}$. The worst-case finishing (ending) time is e_i, e.g., a task which is activated at t_{start} has an absolute completion time $e_i = t_{start} + C_i$ if it is not preempted.

The utilization factor U of a processor under a given set Γ of n independent periodic tasks is defined as $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$. Such a task set is schedulable with EDF under a given frequency f if and only if

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} = \sum_{i=1}^{n} \frac{\check{C}_i \cdot f^{-1}}{T_i} = f^{-1} \sum_{i=1}^{n} \frac{\check{C}_i}{T_i} \leq 1.$$

2.4 Intra-Task Scheduling

Intra-task scheduling aims to save power by lowering the processor frequency and simultaneously by eliminating internal task slack times. Such slack times always occur whenever there is a branching in the control flow of a task and the actual execution path involves less cycles than the worst-case execution path. In the following, we will give a brief overview of intra-task voltage scaling based on [15].

Consider the *Control Flow Graph* (CFG) in Figure 1. Each node represents a basic block of the example code in Figure 2. The nodes include the number of cycles for the execution of the corresponding blocks. The values in brackets indicate the remaining worst-case execution cycles when entering a specific block. Multiple values refer to nodes which can be executed several times, i.e., which are part of a loop.

In the case that condition *cond* is not true, block $b_{if,2}$ will be directly executed after block b_{wh}. If this situation occurs before any execution of the `while`-loop, the remaining worst case execution cycles ($\check{C}_{RWEC}$) after block b_{wh} decrease from $\check{C}_{RWEC}(b_{if,1}) = 425 - 5 = 420$ to $\check{C}_{RWEC}(b_{if,2}) = 75$ cycles. The frequency within the task can be reduced by the scaling factor $\beta(b_{wh} \rightarrow b_{if,2}) = \frac{75}{420}$, because the current frequency was chosen in a way that execution of 420 cycles will be finished in time, but only 75 cycles are needed. Now

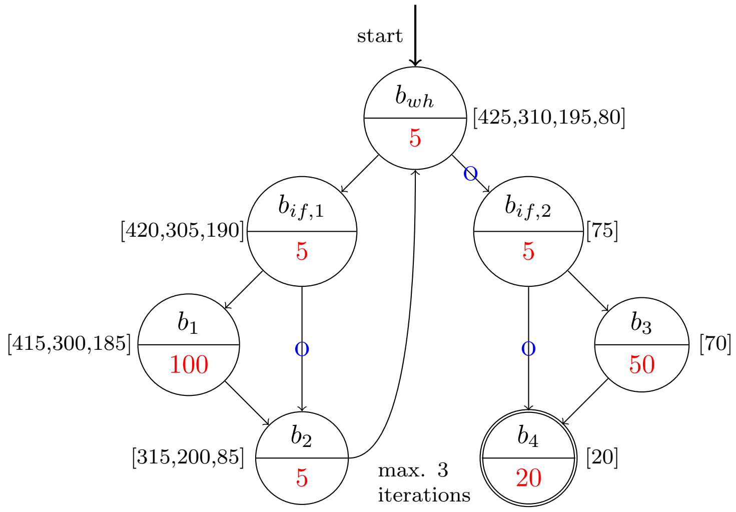


Figure 1: Control Flow Graph (CFG) with timing information: The nodes represent basic blocks including their execution cycles. The values in brackets next to each node give the remaining worst-case execution cycles for each loop iteration.

the processor frequency $S(b_{if,2})$ at block $b_{if,2}$ is updated according to

$$S(b_{if,2}) = S(b_{wh}) \cdot \beta(b_{wh} \rightarrow b_{if,2}).$$

However, this scaling factor changes, if the `while`-loop is executed 1,2 or at most 3 times. As this information is not known in advance, the authors in [15] use so-called L-Type Voltage Scaling Edges (VSE) which calculate the scaling factor $\beta(b_{wh} \rightarrow b_{if,2})$ during run-time depending on the number of loop iterations.

A second type are B-Type VSEs and they are used for `if`-statements (line 8 in our example code). As the number of saved cycles does not depend on loop iterations the scaling factor is $\beta(b_{if,2} \rightarrow b_4) = \frac{20}{70}$. Here, we only need 20 cycles, but remaining 70 cycles were in the budget of the task. Note that this simple scheme for B-Type VSEs can not be used, if we consider branches within loops (line 3, e.g.). Then the number of remaining cycles depends on the iteration count of the outer loop and the scaling factor $\beta(b_{if,1} \rightarrow b_2)$ has to be dynamically computed at run time. (The scaling factor is equal to $\frac{315}{415}$, $\frac{200}{300}$, or $\frac{85}{185}$, depending on whether the loop is iterated for the first, second, or third time). Of course, the computation of the right scaling factor becomes even more involved, when the branch is within a larger number of nested loops.

In contrast, our scheduler accepts relative information about the number of saved cycles in order to simplify frequency scaling. As we will describe in more detail in Section 3.2.1, our scheme not only simplifies the instrumentation of code, but also makes instrumentation of shared code possible. This scheme is used in an overall approach which integrates intra-task frequency scaling and inter-task frequency scaling for preemptive environments with a dynamic number of tasks and a processor model with discrete frequencies.

```
while cond do
    // max. 3 iterations
    if cond1 then
        b_1;
    end if
    b_2;
end while
if cond2 then
    b_3;
end if
b_4;
```

Figure 2: Example code for the Control Flow Graph in Figure 1.

3 Intra- and Inter-Task Frequency Scaling

Our method for real-time scheduling which is presented in this section aims at obtaining a homogeneous distribution of processor frequencies at a level which is as low as possible and thus minimizes energy consumption. In order to guarantee temporal constraints, the utilization of idle times, intra- and inter-task slack times has to be performed without violating deadlines of tasks.

3.1 Idle-time Elimination

For EDF, a real-time schedule contains idle times iff $U < 1$. These idle times can be minimized if a processor is run with a lower frequency f (and indirectly with lower voltage and power consumption) according to a global frequency scaling factor α.

We consider a processor with m discrete frequencies $F = \{f_1, ..., f_m\}$ with $f_i < f_j$ for $1 \leq i < j \leq m$. Let α $(0 < \alpha \leq 1)$ be the scaling factor with respect to the maximum possible frequency f_m.

Schedulability is still guaranteed when scaling all n tasks by a factor α with respect to the maximum possible frequency f_m iff

$$(\alpha \cdot f_m)^{-1} \sum_{i=1}^{n} \frac{\check{C}_i}{T_i} \leq 1.$$

For obtaining highest energy savings we will use the lowest possible α (which results in the lowest frequency) and we get

$$(\alpha \cdot f_m)^{-1} \sum_{i=1}^{n} \frac{\check{C}_i}{T_i} = 1 \Leftrightarrow \alpha = f_m^{-1} \sum_{i=1}^{n} \frac{\check{C}_i}{T_i}.$$

As most processors only support discrete frequencies F, we could use the lowest frequency which is higher than $f_\alpha = \alpha \cdot f_m$ to ensure that for the resulting utilization factor

$U \leq 1$ holds:

$$f'_\alpha = \min\{f_i \in F | f_i \geq \alpha \cdot f_m\}$$

The frequency f is the lowest frequency to run the processor without violating real-time constraints provided that computation cycles $\check{C}_i$ are fixed to their worst-case estimation. However, computation cycles of task execution are not constant – they can be reduced by branching conditions or premature abort of loops during run-time. In that case we can decrease the processor frequency even more in order to execute less cycles in the same time.

3.2 Intra-Task Slack-time Elimination

In Section 2.4, we have given a brief introduction to intra-task frequency scaling. In our work the intra-task frequency scaling is supported by the operating system which takes slack times into account for processors with discrete frequencies. Our approach differs from [15] with respect to three issues:

- We use a simplified scheme for computing remaining worst-case execution cycles. This scheme is based on an estimation of worst case execution cycles for the task as a whole (by static analysis), relative information about saved execution cycles within the control flow, and a cycle counter of the processor. This scheme has the additional advantage that our method is suitable for shared code in libraries used by different tasks.

- The computation of frequencies is not based on scaling factors which take into account the previous estimation for worst case remaining cycles and a new estimation due to saved cycles (as in [15]), but it is changed in order to be able to make use of inter-task slack *time* and of intra-task slack *time* which remain due to a discrete set of processor frequencies.

- The computation of worst-case remaining execution cycles $\check{R}_i$ of a task τ_i, the computation of new processor frequencies, and the according change of frequencies can be seamlessly integrated into an operating system.

3.2.1 Keeping track of remaining worst-case execution cycles

As a basis for frequency rescaling we need an estimation of worst-case remaining execution cycles $\check{R}_i$ of tasks τ_i. As described in Sect. 2.4 the computation of worst-case remaining execution cycles $\check{R}_i$ as given in [15] may need rather involved computations at run-time. Especially if a frequency scaling point is located in several nested loops, an estimation of worst-case remaining execution cycles $\check{R}_i$ will need the current number of iterations for each of the outer loops which were performed so far (in order to be able to compute the worst-case remaining iteration counts based on the respective maximal iteration counts).

Our scheme for computing remaining worst-case execution cycles is much simpler: It starts with an estimation of worst case execution cycles for the task as a whole which is obtained by static analysis. In order to keep track of the remaining worst-case execution cycles $\check{R}_i$, the processor has to provide a cycle counter $\check{C}_{CLK}$ which is incremented during program execution. For each task τ_i, the operating system then maintains a task-specific cycle counter $\check{C}_{act,i}$ derived from $\check{C}_{CLK}$. Based on the initial number of worst case execution cycles for τ_i, the number of cycles used for τ_i, and the number of cycles saved in τ_i (compared to the worst-case estimation), the remaining worst-case execution cycles $\check{R}_i$ are computed. The number of cycles saved in τ_i is updated at frequency scaling points by additional code as will be described below.

In more detail, the counter $\check{C}_{act,i}$ is reset to 0 whenever $\check{R}_i$ is updated (see also Section 3.5). This happens in the following three cases: The first case is when a task τ_i is activated – here, $\check{R}_i$ is initialized to $\check{C}_i$. The second case occurs when a task τ_i is preempted. Assume that τ_i has already executed $\check{C}_{act,i}$ cycles since the last update, then the remaining cycles are updated with

$$\check{R}_i \leftarrow \check{R}_i - \check{C}_{act,i}$$

The third case occurs when information on saved cycles is evaluated during task execution. Here, the remaining execution cycles $\check{R}_i$ are updated at Frequency Scaling Points (FSP).

R-type Frequency Scaling Points We call the Frequency Scaling Points (FSPs) mentioned above R-type FSPs, because they give relative information about saved cycles in the control flow. Consider that the executed task traverses the edge $b_{if,1} \rightarrow b_2$ in Figure 1. Then, the block b_1 would be omitted and 100 cycles are saved. Such relative information is independent of the number of (potentially nested) loop iterations performed so far. Whenever a task τ_i reaches a R-type FSP which saves $\check{C}^R$ cycles, it updates the remaining cycles $\check{R}_i$:

$$\check{R}_i \leftarrow \check{R}_i - \check{C}_{act,i} - \check{C}^R \quad \text{if R-type FSP}$$

It is easy to see that the scheme based on cycle counters and relative information can be implemented with low overhead for computing remaining worst-case execution cycles. Another key advantage consists in the fact that it is *suitable for shared code* in libraries used by different tasks. In this case absolute information about remaining worst-case execution cycles can not be written into the shared code, because the information differs depending on the context in which the shared code is used. Thus, the annotation scheme from [15] is not applicable. In our scheme shared code contains only relative information about saved cycles and the remaining worst-case execution cycles are computed based on this as described above.

A-type Frequency Scaling Points For completeness, we mention a further optimization for special cases within non-shared code. For these cases we introduce A-type FSPs which keep absolute information about remaining worst-case execution cycles. Consider edge $b_{wh} \rightarrow b_{if,2}$ in Figure 1. Here, the number of remaining cycles can directly be determined (75 cycles). An A-type FSP can only be used when the corresponding edge is not located inside a loop or inside shared code. The absolute number of remaining worst-case execution cycles is directly updated by

$$\check{R}_i \leftarrow \check{C}^A \quad \text{if A-type FSP.}$$

A-type FSPs make sense when R-type FSPs do not provide perfect knowledge about saved cycles (e.g., if not all branches in the control flow are annotated with R-type FSPs or if worst-case execution time computation is imprecise due to advanced features of processors like caches and pipelines).[3]

3.2.2 Frequency determination

Now we look into the question how to compute frequencies based on worst case remaining cycles. To be able to use frequency scaling in a multi-task environment together with inter-task frequency scaling we do not use the old estimation for worst case remaining cycles (before saving cycles) as a basis for computing the new frequency, but we use a 'remaining time' $t_{remain,i}$ which depends on inter-task slack-time elimination (see Sect. 3.3) and on effects of intra-task slack-time elimination with discrete frequencies.

The lowest possible frequency to run a single task τ_i with $\check{R}_i$ remaining cycles and a remaining time $t_{remain,i}$ is

$$f = \frac{\check{R}_i}{t_{remain,i}} = \frac{\check{R}_i}{e_i - t}$$

with an absolute finishing time e_i and the current time t. Of course – due to a set of discrete frequencies of the processor – the computed frequency f may not be available. One option would be to use a frequency $f' = \min\{f_i \in F | f_i \geq f\}$ instead. An alternative solution will be described in Section 3.4.

The values of e_i, t and $\check{R}_i$ are provided by the operating system and depend on inter-task slack-time elimination as we will see in Sect. 3.3. The absolute finishing time e_i potentially accounts for inter-task slack-time which will be used here by the intra-task algorithm. Note that also slack time which occurs due to discrete frequency selection at earlier FSPs is accounted for by a smaller value of the current time t and thus is considered in an elegant way early inside the intra-task method.

3.3 Inter-Task Slack-time Elimination

Now we describe our approach to inter-task slack elimination which is deeply integrated with intra-task slack elimination. We will start with an example illustrating the interaction between inter- and intra-task slack elimination.

In Figure 3 we consider a processor with discrete frequencies $F = \{100\,\text{MHz}, 133\frac{1}{3}\,\text{MHz}, 166\frac{2}{3}\,\text{MHz}, 200\,\text{MHz}\}$ and two ready tasks with $2 \cdot 10^5$ remaining execution cycles for the first one and 10^5 cycles for the second one. Moreover, the deadline for the first task is at

[3]For shared code, A-type FSPs can be transformed into R-type FSPs where $\check{C}^R$ is computed depending on the number of saved cycles.

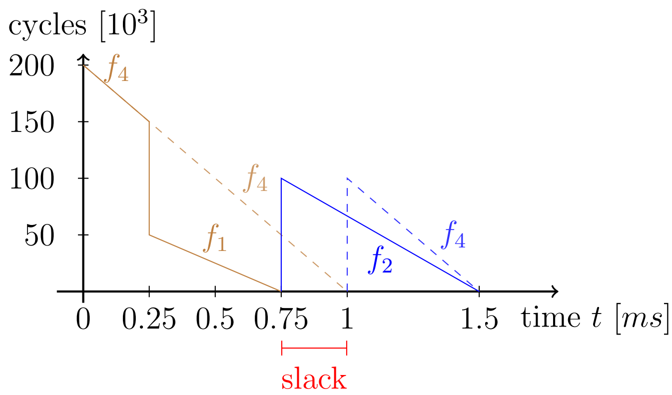


Figure 3: Combined intra- and inter-task scheduling: Additional slack of intra-task frequency scaling on processors with discrete frequencies is passed to the subsequent task

$t = 1$ ms and for the second one at $t = 1.5$ ms. At $t = 0.25$ ms, the first task is rescaled to the lowest possible frequency $f_1 = 100$ MHz due to intra-task information which indicate a decrease of 10^5 cycles and finishes at $t = 0.75$ ms. The resulting slack of 0.25 ms can be used to rescale the subsequent task to $f_2 = 133\frac{1}{3}$ MHz instead of $f_4 = 200$ MHz as the remaining time for execution is 0.75 ms instead of 0.5 ms.

In order to consider such an inter-task slack time we use the same principle as the EDF based frequency scaling algorithm given in [7]. Whenever the actual completion time of a task is less than its worst-case completion time e_i the additional slack time is passed to the subsequent task. The (absolute) worst-case completion time e_i can be calculated incrementally whenever a task τ_i starts, preempts (another task) or resumes at current time t. The computation of e_i makes use of execution times C_i which are reserved for tasks τ_i. C_i results from the worst case execution cycles $\check{C}_i$ by $C_i = \check{C}_i \cdot f_\alpha^{-1}$ where f_α is the frequency determined as described in Sect. 3.1. (If all tasks run with frequency f_α and use their worst-case execution cycles, then the processor utilization U exactly equals 1 and EDF scheduling is able to find a feasible schedule.) The absolute worst-case completion times e_i are computed as follows:

$$e_i = \begin{cases} t + C_i & \text{if } \tau_i \text{ preempts task } \tau_j \\ e_i + e_k - t_i^p & \text{if } \tau_i \text{ was preempted at } t_i^p \\ & \text{and resumes after task } \tau_k \\ e_k + C_i & \text{if } \tau_i \text{ starts after } \tau_k \\ & \text{and } d_i \geq d_k \wedge t < e_k \\ t + C_i & \text{otherwise} \end{cases} \tag{1}$$

In cases 2 and 3 this scheme ensures that a possible slack is used by the following task. The difference of $e_i - t$ gives the available time for task τ_i. Then the lowest possible frequency is

$$f = \frac{\check{C}_i}{e_i - t}.$$

The absolute worst-case completion time e_i is used during intra-task slack-time elimination to further decrease the frequency whenever it detects saved cycles at frequency scaling points.

The correctness of the overall approach follows from the correctness of [7] together with the observation that intra-task frequency scaling never increases the computation time of a task τ_i beyond $C_i = \check{C}_i \cdot (f_\alpha)^{-1}$.

3.4 Split Frequency Rescaling

In [6], Ishihara and Yasuura considered the problem of minimizing energy consumption for variable voltage processors. They proved that the *two voltages which minimize energy consumption under a time constraint are immediate neighbors to the v_{ideal}* – the optimal voltage for a continuous voltage processor. A similar scheme is applied for discrete frequencies in [4], so that a task with a deadline constraint is firstly executed with a lower frequency and later switched to a higher frequency to meet its deadline for the worst-case. In contrast to [4] we also perform splitting at each frequency scaling point. More precisely, we split the time needed to execute the remaining worst-case execution cycles $\check{R} = \check{R}_a + \check{R}_b$ which should be executed with frequency f into two time intervals executed at the largest frequency $f_a \leq f$ and the smallest frequency $f_b \geq f$. For the number $\check{R}_b$ of remaining execution cycles when we have to switch to the higher frequency f_b we conclude from $R = \check{R}_a + \check{R}_b$ and $\check{R} \cdot f^{-1} \geq \check{R}_a \cdot f_a^{-1} + \check{R}_b \cdot f_b^{-1}$ that

$$\check{R}_b = \left\lceil \check{R} \cdot \frac{f^{-1} - f_a^{-1}}{f_b^{-1} - f_a^{-1}} \right\rceil .$$

Therefore, the tasks starts with f_a and, if there are only $\check{R}_b$ cycles left, the scheduler switches to the higher frequency f_b. For a real processor, this frequency switch can be triggered with a previously set timer.

Note that split frequency rescaling aims to eliminate slack times due to discrete frequencies and – in the ideal case – no inter-task slack should occur. However, this goal can not always be achieved as the computed ideal frequency can be lower than the lowest available frequency f_1 and, therefore, a task instance would still produce slack after its completion. Another reason to consider inter-task slack can occur in the application of this approach for complex architectures, e.g. with caches. Firstly, static analysis tools tend to overestimate worst-case execution times. Secondly, a task can not detect all saved cycles as their number depends on the preceding cache accesses.

3.5 Integration into Operating System

In this section we give two algorithms to be be integrated into an operating system. They implement our overall approach based on the concepts for idle, intra- and inter-task slack time elimination. In order to calculate the minimal frequency in our method Algorithm 1 is called whenever a new periodic task arrives ($\Gamma \leftarrow \Gamma \cup \{\tau_i\}$) or is removed from the task set ($\Gamma \leftarrow \Gamma \setminus \{\tau_i\}$). It calculates the utilization factor and determines if the new task set is still feasible. If the new task is accepted, the scaling factor α for idle-time elimination is determined.

Algorithm 1 Task management (creation and removal)

manage_task(τ_i)
  if new periodic task τ_i arrives **then**
    $\alpha' \leftarrow \alpha + f_m^{-1} \frac{\check{C}_i}{T_i}$
    if not($\alpha' \leq 1$) **then**
      return 'not feasible'
    end if
    $\Gamma \leftarrow \Gamma \cup \{\tau_i\}$
    $\alpha \leftarrow \alpha'$
  else if remove task τ_i **then**
    // when task instance finishes
    $\alpha \leftarrow \alpha - f_m^{-1} \frac{\check{C}_i}{T_i}$
    $\Gamma \leftarrow \Gamma \setminus \{\tau_i\}$
  end if
  $f_\alpha \leftarrow f_m \cdot \alpha$
  // update values for all tasks τ_j
  $C_j \leftarrow \check{C}_j \cdot f_\alpha^{-1}$
  $e_j \leftarrow t + C_j$
  $t_j^p \leftarrow t$
  return 'ok'

Algorithm 2 combines the concepts of inter- and intra-task slack time elimination whenever there is a context switch or program execution reaches a FSP. In the main, it updates the values for the remaining cycles $\check{R}_i$ and the absolute finishing time e_i of each task τ_i.

Please note that for each context switch and each execution at an intra-task frequency scaling point the needed computations can be performed in constant time.

3.6 Overhead Considerations

In order to include overhead due to intra-task frequency scaling points and inter-task context switches a fixed number of cycles can be added to the worst-case execution cycles $\check{C}_i$ of a task τ_i as discussed in the following.

For *inter-task overhead* we consider the worst-case number of cycles $\check{C}_{\text{CS}}$ for a context switch. Each context switch is always connected with an activation or termination of a task, i.e., the overhead can be estimated by $2 \cdot \check{C}_{\text{CS}}$ per task instance. This consideration includes a switch between tasks, a switch from and to a (possibly virtual) idle task, and preemption of a task.

Algorithm 2 Called on inter- or intra-task scheduling

$calculate\_frequency(\tau_i)$

  if intra-task scaling of τ_i **then**
    // task τ_i saved some cycles at a FSP
    if absolute FSP **then**
      $\check{R}_i \leftarrow \check{C}^A$
    else
      $\check{R}_i \leftarrow \check{R}_i - \check{C}_{act,i} - \check{C}^R$
    end if
  else if inter-task scaling of τ_i **then**
    if τ_i preempts τ_j **then**
      $e_i \leftarrow t + C_i$
      $\check{R}_i \leftarrow \check{C}_i$
      $\check{R}_j \leftarrow \check{R}_j - \check{C}_{act,j}$ // $\check{C}_{act,j}$ since last switch
    else if r_i resumes after some task τ_k **then**
      $e_i \leftarrow e_i + e_k - t_i^p$ // τ_i was preempted at t_i^p
    else
      // τ_i starts execution after some task τ_k has finished
      if $d_i \geq d_k$ and $t < e_k$ **then**
        $e_i \leftarrow e_k + C_i$
      else
        $e_i \leftarrow t + C_i$
      end if
      $\check{R}_i \leftarrow \check{C}_i$
    end if
  end if
  $\check{C}_{act,i} \leftarrow 0$
  $f \leftarrow \frac{\check{R}_i}{e_i - t}$
  determine $\check{R}_{i,b}$ with next lower frequency f_a and higher frequency f_b and set timer which switches to frequency f_b when $\check{R}_{i,b}$ cycles are left
  $set\_frequency(f_a)$

Hence, an upper bound for the additional system utilization U_{CS} can be determined by

$$U_{\text{CS}} = \frac{2\check{C}_{\text{CS}}}{f_{\text{CS}}} \sum_{i=1}^{n} \frac{1}{T_i}$$

assuming that scheduler routines are executed with a frequency of at least f_{CS}.

Of course, tools for WCET estimation are also able to analyze code with intra-task instrumentation and thus, additional cycles for *intra-task overhead* due to frequency scaling points can be determined. However, only conditional branches in the CFG where the overhead for processing a FSP is smaller than the number of saved cycles will be instrumented, of course. This will lead to an incomplete instrumentation in practical applications.

4 Experiments

For our experiments we implemented a simulation framework which determines the energy efficiency with different task sets and hardware configurations. In order to evaluate the effectiveness of our approach we implemented various well-known DVS schedulers, e.g., LaEDF [10] and OLDVS [7], and compared the results to our scheduler implementation ItcaEDF (*Intra-Task Characteristics Aware EDF*). Before we discuss the results in Section 4.3 we give a more detailed description of our simulation framework and system setup.

4.1 Simulation Framework

Our experiments were performed with a compiler and simulation framework written in C++. As our approach utilizes intra-task slack due to unused execution paths during program execution it was not only necessary to model task execution times but also their control flow. To achieve this, task behavior is described in a language based on ANSI C in order to define task behavior including control flow, task semantics and temporal characteristics. Syntax extensions provide the possibility to specify execution times for statements and to annotate code with *flow facts* like upper bounds for loops. These extensions are used for both integrated path-based WCET analysis and simulation. Finally, in our framework we implement frequency scaling points with a code instrumentation scheme, i.e., after path-based WCET analysis we instrument the task with additional statements giving hints about the number of saved cycles $\check{C}^R$. The number of saved cycles at a FSP is constant for conditional branches in the CFG. For loops, it is dynamically computed depending on the worst-case numbers of iterations and the actual numbers of iterations.

Moreover, we implemented the concepts of our approach in a prototype micro-kernel for the Atmel ATMega128 micro-controller. The execution of the micro-kernel and their tasks were performed in a simulation and analysis framework for Atmel micro-controllers. Whereas in this paper we report on results obtained using the simulation framework described above, these results are corroborated by first measurements in the context of our prototype micro-kernel.

4.2 System and Task Setup

In our simulations we use a processor model with four different operating configurations $S = \{(250\,\text{kHz}, 2\,\text{V}), (500\,\text{kHz}, 3\,\text{V}), (750\,\text{kHz}, 4\,\text{V}), (1\,\text{MHz}, 5\,\text{V})\}$. The energy consumption is computed assuming each cycle needs an energy amount proportional to the square of the operating voltage. As we are only interested in the relative energy consumption of the DVS schedulers we assume that idling the processor consumes no energy as discussed in [10]. In the following we will provide details on how we generated task sets in our implementation.

Firstly, we choose an overall worst-case utilization factor $U \leq 1$. This utilization factor is randomly split under the given number of tasks, so that $U = \sum_{\tau_i \in \Gamma} U_i$ and $\forall \tau_i, \tau_j : \frac{1}{l} \cdot U_j \leq U_i \leq l \cdot U_j$ for a given $l \in \mathbb{R}$. We choose $l = 2$ to avoid exceeding differences in task characteristics. Subsequently, the system determines a period $T_i \in [100\,\text{ms}, 1000\,\text{ms}]$ and computes the worst-case execution time according to $C_i = U_i \cdot T_i$. Finally, this worst-case execution time is realized by a nested loop:

for $i = 1, .., i_{\max} = b_{\text{outer}}$ **do**
  // randomly determine $j_{\max} \in [b_{\text{inner,min}}, b_{\text{inner,max}}]$
  // automatically inserted intra-task frequency scaling point
  for $j = 1, .., j_{\max} \leq b_{\text{inner}}$ **do**
    // computation with fixed number of cycles $\check{C}_{i,\text{loop}}$
  end for
end for

The real worst case numbers of cycles $\check{C}_{i,\text{loop}}$ needed in the nested loop and the final overall worst-case utilization factor were computed so that $C_i \cdot f_m = b_{\text{inner}} \cdot b_{\text{outer}} \cdot \check{C}_{i,\text{loop}}$.

In this paper we specifically set $b_{\text{outer}} = 5$ and $b_{\text{inner}} = 10$, therefore the inner loop including the FSP is executed five times. The values of $b_{\text{inner,min}}$ and $b_{\text{inner,max}}$ were varied to realize different ranges for the actual execution time (AET) to worst-case execution time (WCET) ratio. For example, values of $b_{\text{inner,min}} = 4$ and $b_{\text{inner,max}} = 8$ can be used to force an average fraction of 0.6 for the actual workload of a task.

4.3 Results

The simulation results are given in Figure 4 for different task scenarios. Each diagram shows a histogram for the energy efficiency of a specific scheduler. On the x-axis we have the worst-case utilization factor of the corresponding task set and on the y-axis the energy consumption for the different methods as a fraction of the energy consumption for an EDF based execution with the highest possible frequency f_m. In the experiments we varied the number of executed tasks n and the 'actual workload'. The 'actual workload' denotes the fraction c of the actual execution time (AET) to the worst-case execution time (WCET) of a task as described above. The diagrams in this paper refer to scheduling with either $n = 2$ or $n = 8$ tasks and an actual workload either in the interval $[0.8, 1.0]$ or $[0.4, 0.8]$.

We separated the results for the different schedulers into two blocks, respectively. Before we discuss the results we give a short overview of the compared energy-efficient scheduling methods. The first block consists of real-time DVS schedulers known from literature:

- **StaticEDF:** StaticEDF uses a constant frequency depending on the worst-case utilization U of the task set.
- **OrigIntra**: Original Intra-Task DVS which uses fixed ratios for frequency rescaling [15].
- **OLDVS**: Slack passing scheme which gives unused computation to the subsequent task [7].
- **OLDVS*:** Improved OLDVS variant for discrete frequency processors which splits execution time in two parts executed with the next lower and next higher discrete frequency [4].
- **LaEDF**: Speculative Look-ahead EDF scheduler which scales to the lowest possible frequency by deferring as much work as possible after the next deadline [10].

The second block consists of inter-task schedulers which include intra-task voltage scaling according to the scheme presented in this paper. Here, the last method *ItcaEDF* is our proposed approach which outperforms all other DVS schedulers:

- **IntraLaEDF**: Variant of LaEDF combined with our intra-task approach. Rescaling with deferring work after the next deadline is done at each frequency scaling point and at each context switch.
- **IntraOLDVS**: OLDVS combined with our intra-task approach.
- **ItcaEDF (Intra-Task Characteristics Aware EDF)**: The proposed on-line algorithm which integrates our intra-task approach and split frequency rescaling (see Sect. 3.4).

In all configurations the minimal energy consumption depends on the lowest available frequency. Therefore for utilization factors smaller than the ratio $f_1/f_m = 0.25$ all frequency scaling techniques gave the same energy consumption as all tasks already start with the lowest possible frequency. The energy consumption increases with higher worst-case utilization factors. As expected, StaticEDF shows a stepwise increase of energy consumption (according to the discrete frequency distribution), because only worst-case execution times and no occurring slack is considered. (Energy consumptions for $U = 0.10, 0.20$, $U = 0.30, 0.40, 0.50$, $U = 0.60, 0.70$, and $U = 0.80, 0.90$, respectively, are identical.)

Considering the first block of schedulers known from literature neither OrigIntra nor OLDVS show an overall good performance. OrigIntra performs comparatively well when the actual workload is low (see second and fourth histogram), since it is able to rescale with a high granularity at task-internal frequency scaling points. However, OrigIntra shows high energy consumption when the actual workload is large (i.e., not far away from the worst case workload, see first and third histogram) as the occurring slack for a single task is too small in order to scale to a lower frequency and slack is not accumulated between tasks (since it is not passed to a subsequent task). OLDVS shows only moderate savings compared to StaticEDF, since frequency scaling with slack passing to the subsequent task is performed only at context switches with a lower granularity. The original OLDVS scheduler is improved by the split frequency scaling of OLDVS* which provides a better support for discrete frequencies on the one hand and on the other hand works more optimistically by selecting the lower of the two neighbor frequencies first. The most optimistic (or in other words most aggressive) method is LaEDF which works especially well when the actual workload is low and the number of tasks is high (see fourth diagram). However, when the workload is too high, the speculative scheme of LaEDF (which defers as much work as possible after the next deadline) seems to be too aggressive and the likelihood that, for example, a preempting task has to choose a higher frequency is increased (see first and third diagram). When the number of tasks is low (see first and second diagram) LaEDF also may suffer from the fact that the number of context switches becomes smaller and there are less opportunities for frequency scalings.

Considering the last blocks, our proposed ItcaEDF approach clearly outperforms the other schedulers in all task setups. For instance, considering a worst-case utilization of $U = 0.8$, a AET/WCET ratio between 0.4 and 0.8, and two tasks (see diagram 2), ItcaEDF improved the normalized energy consumption by 28% compared to OLDVS* and 34% compared to LaEDF. Increasing the number of tasks to 8 (see diagram 4) results in a energy reduction of 16% compared to LaEDF and 31% compared to OLDVS*. Apparently, in our scheduler the deep integration of intra-task code instrumentation into inter-task frequency scaling really pays off. The intra-task frequency scaling points both provide timing information to reflect the actual work load of the system more accurately and give a higher granularity for efficient scheduling with DVS. ItcaEDF avoids to work too aggressive (as LaEDF in some cases), since the only case of an optimistic frequency selection occurs when a not existing continuous frequency is approximated by two discrete frequencies and the lower one is selected first. (However this decision is confirmed by the observation that the need for using the second (higher) frequency is often canceled in the future, when the actual workload is low and additional slack occurs after the frequency splitting.)

For completeness, we also considered schedulers where our intra-task code instrumentation scheme is integrated into OLDVS directly (without frequency splitting) and into LaEDF (the corresponding bars in the histograms are labeled IntraOLDVS and IntraLaEDF, respectively). As expected, IntraOLDVS (in comparison to OLDVS) profits from the detection of reduced remaining worst-case execution cycles during task run-time, but it is outperformed by ItcaEDF. Somewhat surprisingly, the results for IntraLaEDF show that it is not always a good idea to combine an inter-task scheduler with intra-task code instrumentation: LaEDF can hardly benefit from our code instrumentation approach and IntraLaEDF shows sometimes even a higher energy consumption than LaEDF. Obviously, increasing the aggressiveness of the speculative algorithm by exploiting intra-task characteristics has a negative impact on energy-efficiency.

5 Conclusions

In this paper, we proposed an energy-efficient real-time scheduler which incorporates cycle-based intra-task and time-based inter-task frequency scaling for the realistic scenario that we have processors with discrete frequencies and dynamic task sets. A multi-level approach considers idle, intra- and inter-task slack times and scales the processor speed to the lowest possible frequency on context switches and within task execution. We also presented a novel technique to keep track of remaining worst-case execution cycles for intra-task frequency scaling which is based on cycle counters of the hardware and which is suitable for shared code. To evaluate the performance of the on-line scheduler we integrated the algorithms into a compiler and simulation framework. Our experimental results showed that our approach is able to reduce energy consumption of state-of-the-art inter-task DVS schedulers by over 30%.

Acknowledgement

This work has partly been supported by the German Research Foundation (DFG) within the Research Training Group 1103.

References

[1] Hakan Aydin, Rami Melhem, Daniel Mossé, and Pedro Mejía-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Trans. Comput.*, 53(5):584–600, 2004.

[2] T. D. Burd and R. W. Brodersen. Energy efficient CMOS microprocessor design. In *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, page 288, Washington, DC, USA, 1995. IEEE Computer Society.

[3] A.P. Chandrakasan, S. Sheng, and R.W. Brodersen. Low-power CMOS digital design. *Solid-State Circuits, IEEE Journal of*, 27:473–484, 1992.

[4] Min-Sik Gong, Yeong Rak Seong, and Cheol-Hoon Lee. On-line dynamic voltage scaling on processor with discrete frequency and voltage levels. In *ICCIT '07: Proceedings of the 2007 International Conference on Convergence Information Technology*, pages 1824–1831, Washington, DC, USA, 2007. IEEE Computer Society.

[5] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power CPU. In *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25, New York, NY, USA, 1995. ACM.

[6] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *ISLPED '98: Proceedings of the 1998 international*

symposium on Low power electronics and design, pages 197–202, New York, NY, USA, 1998. ACM.

[7] Cheol-Hoon Lee and Kang G. Shin. On-line dynamic voltage scaling for hard real-time systems using the EDF algorithm. In *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 319–327, Washington, DC, USA, 2004. IEEE Computer Society.

[8] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[9] T. Pering and R. Broderson. Energy efficient voltage scheduling for real-time operating systems. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium RTAS'98, Work in Progress Session*, June 1998.

[10] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102, New York, NY, USA, 2001. ACM.

[11] Claudio Scordino and Giuseppe Lipari. A resource reservation algorithm for power-aware scheduling of periodic and aperiodic real-time tasks. *IEEE Trans. Comput.*, 55(12):1509–1522, 2006.

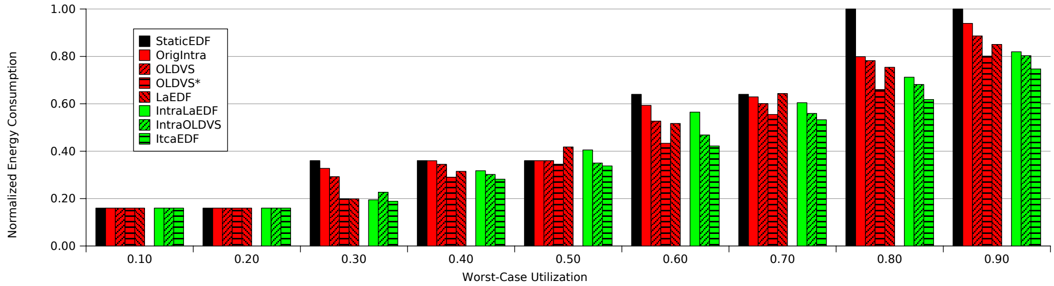
[12] Jaewon Seo, Taewhan Kim, and N. D. Dutt. Optimal integration of inter-task and intra-task dynamic voltage scaling techniques for hard real-time applications. In *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 450–455, Washington, DC, USA, 2005. IEEE Computer Society.

[13] D. Shin and J. Kim. Optimizing intratask voltage scheduling using profile and data-flow information. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26:369–385, 2007.

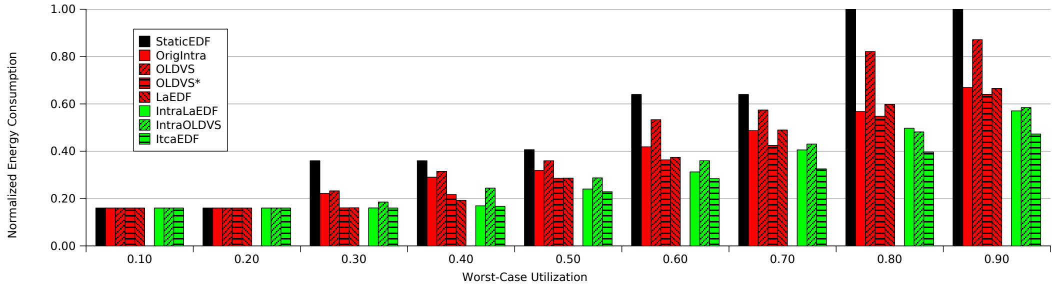
[14] Dongkun Shin and Jihong Kim. Optimizing intra-task voltage scheduling using data flow analysis. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 703–708, New York, NY, USA, 2005. ACM.

[15] Dongkun Shin, Jihong Kim, and Seongsoo Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *Design & Test of Computers, IEEE*, 18:20–30, 2001.

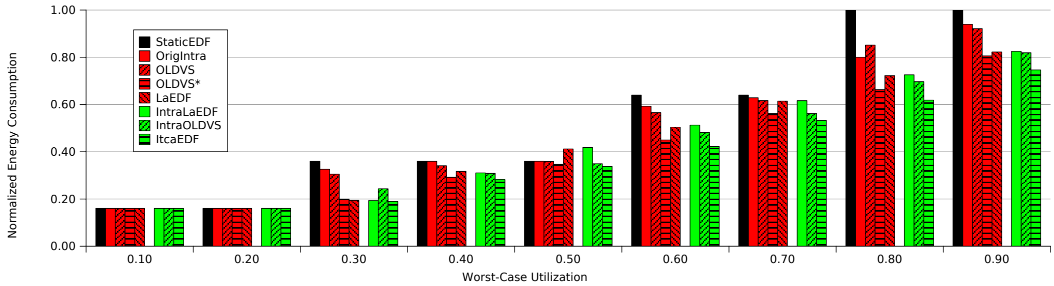
[16] Yifan Zhu and Frank Mueller. Feedback EDF scheduling of real-time tasks exploiting dynamic voltage scaling. *Real-Time Syst.*, 31:33–63, 2005.



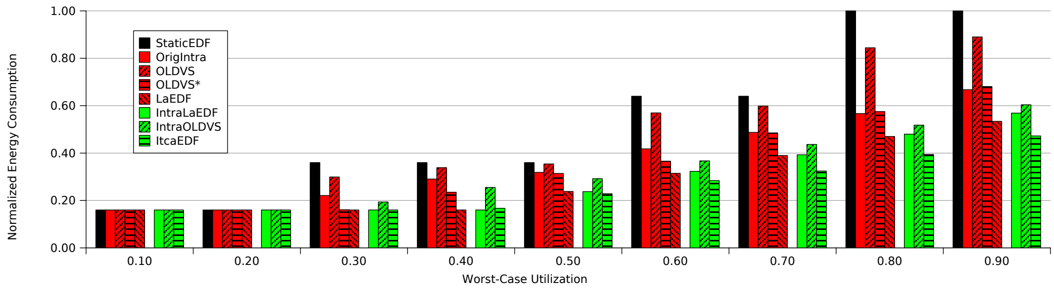
2 tasks, fraction of actual to worst-case execution time between 0.8...1.0 for each task



2 tasks, fraction of actual to worst-case execution time between 0.4...0.8 for each tasks



8 tasks, fraction of actual to worst-case execution time between 0.8...1.0 for each task



8 tasks, fraction of actual to worst-case execution time between 0.4...0.8 for each task

Figure 4: Normalized energy consumption under various task characteristics for 2 or 8 tasks and different fractions of utilization