Sustainable Computing: Informatics and Systems xxx (2012) xxx–xxx

Contents lists available at SciVerse ScienceDirect

Sustainable Computing: Informatics and Systems

journal homepage: www.elsevier.com/locate/suscom

Energy-aware dynamic slack allocation for real-time multitasking systems

Weixun Wang*, Sanjay Ranka, Prabhat Mishra

Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA

ARTICLE INFO

Article history:
Received 7 November 2011
Accepted 30 April 2012

Keywords:
Embedded systems
Real-time systems
Energy optimization
Energy-aware scheduling

ABSTRACT

Dynamic voltage scaling (DVS) has been a very effective technique for processor energy reduction. It adjusts processor voltage and frequency level during runtime. In this article, we propose a general and flexible processor voltage scaling algorithm for real-time multitasking systems. Our approach focuses on exploiting dynamic slack that is created when a task finishes earlier than its estimated worst-case execution time (WCET). Our algorithm is efficient enough to execute at runtime and can be configured flexibly to make tradeoffs between running time and energy savings. By rescheduling tasks effectively, we can achieve almost as much energy savings as if there is no arrival time constraints. Furthermore, our approach can effectively incorporate both leakage power consumption as well as variable scaling overhead. Also, it is relatively independent of task characteristics and scheduling policy. Experimental results show that our technique can achieve significant energy savings at runtime over statically generated schedules and up to 12% more savings compared to the state-of-art techniques.

1. Introduction

Power management and energy conservation are important considerations in improving battery life of laptops as well as embedded systems. Processor consumes the maximum amount of power. Hence, it is the main target for energy optimization. Dynamic voltage scaling (DVS) [1] is acknowledged as one of the most effective processor energy saving techniques. Effectively a linear decrease in supply voltage leads to quadratic power reduction with only linear performance slowdown. Many processors nowadays, both in general-purpose and embedded systems domain, support DVS with multiple voltage levels [2–4]. The overall power dissipation consists of both dynamic energy and static energy. The former, which is consumed by circuit switching, still occupies a major part of the overall consumption and is controlled effectively by using DVS schemes. The latter, which is also known as leakage energy, is getting larger with the technology scaling process [5,6] and is inversely related to the supply voltage. The impact of leakage current and corresponding energy dissipation [7] is that the benefits of DVS is limited or negative below a threshold voltage.

Multitasking systems schedule multiple tasks on a system simultaneously based on provided policies. In real-time systems, each task may have timing constraints in the form of earliest start time and deadline that must be satisfied in order to guarantee system correctness and safety. Earliest Deadline First (EDF) [8] and Rate Monotonic (RM) [9] are the two most widely used scheduling algorithms for real-time systems. EDF and RM use dynamic and static priority based policies, respectively. Both of them require known task set characteristics including worst-case computation-time and inter-arrival periods. In other words, they statically determine the feasible schedules of the given task set.

DVS determines under which processor voltage level each task is executed. Such decisions need to be made carefully in order to minimize overall energy consumption while guarantee all the timing constraints. DVS stretches the clock cycle length (thus leading to increased task execution time) resulting in energy reduction whenever slack (free runtime) is available. *Static slack* is determined based on the Worst Case Execution Time (WCET) of each task. It is analyzed and exploited during the off-line scheduling process. However, in many cases, task's execution time may vary and thus complete earlier than expected at runtime. This could be caused by different input parameter values, environmental conditions, variable execution paths or mix of the above. *Dynamic slack* created due to early completed tasks can be exploited to further reduce the power dissipation of subsequent tasks and is the main focus of this paper.

The main contribution of this article can be summarized as follows. We develop a dynamic slack reclamation (DSR) algorithm for energy-aware scheduling in uniprocessor multitasking systems with the following innovative properties.

1 Our approach iteratively considers multiple tasks for utilizing the dynamic slack available along with necessary task rescheduling. This leads to higher energy savings compared to existing

* Corresponding author. Tel.: +1 3528714925.
E-mail addresses: wewang@cise.ufl.edu (W. Wang), ranka@cise.ufl.edu (S. Ranka), prabhat@cise.ufl.edu (P. Mishra).

2210-5379/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.suscom.2012.04.001

techniques (e.g. [10]), especially when tasks have different power characteristics [11].

2 Our approach can be parameterized to limit the search space of tasks to be considered for slack allocation. This effectively allows tradeoffs between energy saving versus runtime overhead.

Furthermore, our approach is relatively independent of the system characteristics and scheduling policy. It works, for example, either with or without earliest start time constraints. It can also incorporate scaling overhead if necessary. Extensive experimental results show that our technique can achieve significant reduction in energy requirements as compared to only using static scheduling. It also outperforms existing techniques for dynamic slack allocation by 2–12%.

The rest of the paper is organized as follows. Section 2 describes related research works. System model and energy models are introduced in Section 3. Section 4 presents our proposed algorithm. Experimental results are provided in Section 5. We conclude in Section 6.

2. Related work

A large body of research exists for applying static DVS techniques in real-time systems. Early work by Yao et al. [12] described an off-line algorithm for a simplified model. Several variations and extensions have been recently considered for periodic task sets [13] and aperiodic tasks [14]. Task scheduling with and without preemption is considered in [15,16], respectively. Voltage scaling that limits all instances of the task to a single voltage (inter-task DVS) is considered in [17,15,18]. In contrast, intra-task DVS adjusts voltage level multiple times during each task execution [19,20] based on the information collected at runtime. Zhong et al. [15] solved an overall energy minimization problem with consideration of other system components. Since applying static DVS scheduling in real-time systems is a NP-hard problem [18], approximation algorithms are proposed both in inter-task manner [15,18] and preemptive manner [21]. Recent work also incorporates leakage power awareness [22,23,10,24,25].

Dynamic slack reclamation techniques are proposed in [26–28,16,10]. Aydin et al. [26] presented an online algorithm for utilizing unused task running time and a more aggressive speculative mechanism based on expected workload. Pillai et al. [27] adjusted the processor voltage on each job arrival based on the current system utilization or future task's WCET. Kim et al. [28] considered this problem on an ideal continuously scalable processor. Jejurikar et al. [16] presented an algorithm for non-preemptive task sets. However, all these techniques are either based on certain assumptions or for dynamic energy minimization only. Furthermore, they did not consider various energy saving potentials across different tasks which our approach takes advantage of to utilize the slacks more efficiently.

Leakage-aware dynamic slack reclamation technique is proposed in [10]. It is based on the theorem proved in [26] that every task instance can fully reclaim slacks with higher or equal scheduling priority. In their algorithm, a priority queue is maintained for dynamic slacks generated and each newly arrival task simply fetches all the eligible slacks and scales down the voltage level until the critical speed[1] is reached, which is then followed by procrastination. Our approach iteratively assigns the slack to multiple subsequent tasks based on voltage assignments that lead to higher energy savings. The number of subsequent tasks considered are decided by a user defined window. Also, we adjust the voltage level multiple times within each task instance (i.e. job) and carefully allow task rescheduling to make more benefit from the available slack. This leads to an extensive and flexible approach and is shown to lower the energy requirements as compared to [10] with a low runtime overhead.

3. Preliminaries

3.1. System model

The system can be described as:

- A set of m independent tasks $\mathcal{T}\{\tau_1, \tau_2, \ldots, \tau_m\}$ with each task $\tau_i \in \mathcal{T}$ having known attributes including deadline, arrival time, period (if it is periodic) and WCET.
- A uniprocessor task scheduler.
- A voltage scalable processor which supports h voltage levels $\mathcal{V}\{v_1, v_2, \ldots, v_h\}$.

Given the scheduler and task set information, we can determine the original schedule assuming every job requires its WCET. Furthermore, we assume that static slack allocation has been done a priori. Any existing WCET analysis technique and static energy-aware scheduling algorithms can be employed for this purpose. Our goal is to make voltage scaling decisions during runtime whenever a job finishes earlier than its WCET so that the energy consumption of subsequent jobs can be further reduced without violating any timing constraint. The actual execution time (ACET) varies for each task instance during runtime. To quantify and cap the scope of ACET, the best case execution time (BCET) is defined as the lower bound on ACET for each task. Note that WCET, ACET and BCET are all normalized based on the highest voltage level.

3.2. Energy model

Our analytical energy model is adapted from [30], whose accuracy is verified with SPICE simulation, with the extension that various power characteristics are considered. The dynamic power dissipation can be calculated as:

$$P_{\mu P}^{dyn} = C_{eff} \cdot V_{dd}^2 \cdot f \tag{1}$$

where C_{eff} is the processor's effective switching capacitance, V_{dd} is the supply voltage and f is the operation frequency. Here, C_{eff} depends on both the total capacitance C as well as the actual switching activity $\mathcal{K}$. Specifically, in practice, $\mathcal{K}$ is reflected by the running task's activity characteristic, including memory reference locality, bus access frequency and actual chip units used [11]. Hence, we let $C_{eff} = C_{total} \cdot \mathcal{K}$. The threshold voltage V_{th} can be computed as:

$$V_{th} = V_{th1} - K_1 \cdot V_{dd} - K_2 \cdot V_{bs} \tag{2}$$

where V_{th1}, K_1, K_2 are all constants and V_{bs} represents the body bias voltage. Since static current mainly consists of the subthreshold current I_{subth} and the reverse bias junction current I_j, static power is given by:

$$P_{\mu P}^{sta} = L_g \cdot (V_{dd} \cdot I_{subth} + |V_{bs}| \cdot I_j) \tag{3}$$

where L_g denotes the number of devices in the processor circuit, I_j is approximated as a constant and I_{subth} can be calculated by:

$$I_{subth} = K_3 \cdot e^{K_4 V_{dd}} \cdot e^{K_5 V_{bs}} \tag{4}$$

where K_3, K_4 and K_5 are constant parameters. V_{bs} is set to be constrained (e.g. between 0 and −1 V) in order to prevent junction leakage power overriding the gain in lowering I_{subth}. Specifically, we valuate $V_{bs} = -0.7\,V$ [23]. If $P_{\mu P}^{on}$ represents the intrinsic energy

[1] Critical speed is the point lower than which the total energy per cycle will start increasing rather than decreasing [10,29].

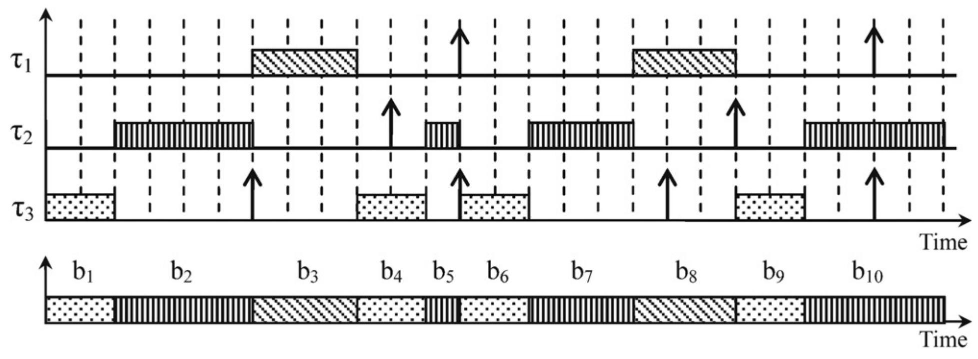


Fig. 1. Execution blocks after static slack allocation.

Table 1
Constants for 70 nm technology.

Const	Value	Const	Value	Const	Value
K_1	0.063	K_6	5.26×10^{-12}	V_{th1}	0.244
K_2	0.153	K_7	-0.144	I_j	4.80×10^{-10}
K_3	5.38×10^{-7}	V_{dd}	[0.5, 1.0]	C_{total}	0.43×10^{-9}
K_4	1.83	V_{bs}	[− 1.0, 0.0]	L_d	37
K_5	4.19	α	1.5	L_g	4×10^6

needed for keeping the processor on (idle energy), the processor power consumption $P_{\mu P}$ can be computed as:

$$P_{\mu P} = P^{dyn}_{\mu P} + P^{sta}_{\mu P} + P^{on}_{\mu P} \quad (5)$$

Frequency f is given by a modified alpha power model:

$$f = \frac{(V_{dd} - V_{th})^{\alpha}}{L_d \cdot K_6} \quad (6)$$

where K_6 is a constant. L_d is estimated to be the average logic depth of all instructions' critical path in the processor. Table 1 lists the constants for a processor with 70 nm technology [30].

Therefore, the processor energy consumption becomes:

$$E_{\mu P} = \frac{P_{\mu P} \cdot n_{cycles}}{f} \quad (7)$$

where n_{cycles} denote the number of clock cycles executed. Note that the same energy model is also used in existing works including [23,29]. Since reducing V_{dd} will lead to increasing of both I_{subth} and I_j, the implication of the above model to handle leakage power is to keep the minimum voltage level (and effectively limit the slowdown) above a threshold, as further voltage reduction will increase the overall energy consumption. Such threshold is called "critical speed" and the related study can be found in our recent work [31].

4. Dynamic slack reclamation

Energy optimization techniques dedicated to static slack allocation derive a scheduling scheme which minimizes energy consumption while guaranteeing all task deadlines. If we execute the tasks under this scheme assuming each of them requires its worst-case workload and let the scheduler record the execution trace, we can get a series of *execution blocks* each of which is a piece of task execution. Note that this step takes during design time. For example, in Fig. 1, we show a preemptive schedule of three periodic tasks.[2] The execution blocks are linearly indexed, e.g. $b_1, b_2, \ldots, b_{10}$ in Fig. 1. Specifically, the input to our problem is represented as:

- A set of n execution blocks $\mathcal{B}\{b_1, b_2, \ldots, b_n\}$. Each block is associated with its corresponding task id and job id.
- Each block $b_i \in \mathcal{B}$ has its arrival time (earliest start time) a_i if it is the first block in the corresponding job and an absolute deadline constraint d_i if it is the last block.
- Each block b_i has its current voltage assignment (thus start time and finish time) after applying the static slack allocation.
- Each block b_i has execution time t_i^k and energy consumption e_i^k at processor voltage level $v_k \in \mathcal{V}$ in the worst-case scenario.

As part of the static analysis, we calculate t_i^k and e_i^k for $\forall i \in [1, n]$ and $\forall k \in [1, h]$ based on either the existing processor datasheets or the energy model described in Section 3.2. We store all the entries for each block with t_i^k lower than the execution time corresponding to its critical speed in a *profile table* with an increasing order of t_i^k (thus decreasing order of e_i^k). In other words, non-beneficial voltage levels are eliminated so that the increase in leakage energy will not compromise the reduction in dynamic energy consumption. Note that varying task's power characteristics lead to different critical speeds. This information is exploited at runtime by our algorithm.

As discussed in Section 1, during actual execution, task instances may take less dynamic instructions (hence shorter time) to complete than the worst-case scenario. The difference between ACET and WCET hence is the generated dynamic slack, as shown in Fig. 2 where the first job of task τ_2 (b_2) finishes earlier by 3 time units. Note that if one job consists of multiple blocks due to preemption, its earlier completion can result in multiple discrete pieces of dynamic slack.

To reclaim dynamic slack, we reassign the voltage levels of one or more subsequent blocks after the slack at runtime. We define *exploration window* as the range of subsequent execution blocks from which the targets of slack reclamation are selected. In other words, we look forward within the exploration window and try to allocate the generated dynamic slack to these tasks in the most beneficial way. Let w denote the size of the exploration window. Clearly, since different blocks may have variable potential for energy reduction (based on the power characteristic and current voltage level assignment), larger w should generally result in better solution but introduce longer time overhead.

[2] Although the example shown in this section is for a preemptive periodic task set, our approach is applicable to other kinds of tasks as long as the characteristics are known a priori and thus the static slack allocated schedule is pre-determined.

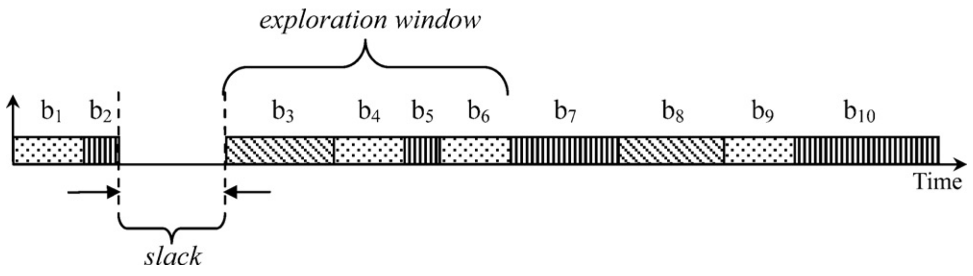


Fig. 2. Dynamic slack generated by early finished task.

There are several design considerations which lead to several variations and finally the full description of our algorithm: (1) whether the tasks have earliest start time (or arrival time) constraints and (2) whether the preemption schedule of a task is allowed to be modified at runtime (i.e. decomposition/agglomeration of the execution blocks). We describe each of these variations in the following sections.

4.1. *Tasks without arrival time constraints*

In order to lower some subsequent tasks' voltage level (i.e. stretch their execution time) to reduce energy requirements, they have to be able to start earlier by the same amount of time. The basic idea of our algorithm is to bring forward (start earlier) every block which receives slack by the difference between the execution time of its previous and new voltage assignments. By doing this, we ensure that no block (in the exploration window) after dynamic slack reclamation finishes later than before. Otherwise, deadline constraints may be violated in the future since it is always possible that all subsequent jobs finish in their WCET. Consider the case when there is no arrival time constraint (Fig. 2). If b_4 and b_6 are selected to be assigned the dynamic slack, b_4 and b_6 as well as all the blocks between them and the one which creates the slack (b_2), which are b_3 and b_5 in this case, should be started earlier as illustrated in Fig. 3. Clearly, no deadline will be violated since we ensure no block in the exploration window gets its completion delayed. Note that when making the decision, we assume that b_4 and b_6 still require WCET to complete. However, they may also finish earlier and create additional slacks later.

Clearly, in the scenario where there is no arrival time constraint (e.g. all tasks are ready when the system begins), it is allowed to freely make any subsequent block start earlier to assign the slack within the exploration window. In other words, all the blocks within exploration window have equal opportunity to take the advantage of reclaiming full amount of slack. Algorithm for assigning the slack will be described in Section 4.3.

4.2. *Tasks with arrival time constraints*

When tasks have arrival time constraints, e.g. periodic tasks, we may not have the freedom to start the execution of a subsequent block earlier to fully reclaim the slacks, i.e., it is possible that not all the blocks within the exploration window have the same capability to receive the slack. In the example shown in Fig. 4, if b_2 finishes earlier to create 3 units of time slack, b_5, unlike b_3 and b_4, is not able to receive the full benefit since it can only be started earlier by at most 1 time unit. Similarly, b_6 cannot be further slowed down without affecting subsequent tasks since it starts right at its arrival time. We define the term *maximum reclaimable slack* (*MaxRS*) for each block as the maximum amount of available slack it can exploit. In this example, b_3 and b_4 have *MaxRS* of 3 units but b_5 has only 1 unit. This observation leads to two variations of our approach.

4.2.1. *Without task rescheduling*

As discussed above, in order to let one block start earlier, all the preceding blocks should also be moved up by the same amount of time. Therefore, within the exploration window, every block's *MaxRS* is no more than any of its predecessors. If it is not allowed to change the original schedule (i.e. block execution order), once a block *B*'s *MaxRS* gets reduced and becomes lower than its precedent block, all the blocks after *B* will also have their *MaxRS* reduced to the same amount. In other words, even if some subsequent blocks can be moved up by the extent more than *B* can, they will still end up with their *MaxRS* at most equal to *B*'s since they can only start after *B* finishes. For example, in Fig. 4, b_6 and all the subsequent blocks are not capable of using any dynamic slack since none of them can start earlier without rescheduling.

4.2.2. *With task rescheduling*

We can prevent the *MaxRS* of block b_i ($MaxRS_i$) from being reduced due to arrival time constraints by changing the task execution order. It is beneficial since it can increase the number of eligible blocks that can receive more slack in the exploration window. By doing this, potentially more energy savings can be achieved. This can be done by bringing forward the execution of some subsequent blocks (or part of them), say b_j where $j > i$, before b_i. As illustrated in Fig. 5, for example, some block before b_1 finishes earlier and creates a piece of slack with length *s*. While we have $MaxRS_1 = s$, however, b_2 is not able to take any advantage since it starts at its arrival time and thus cannot be moved up ($MaxRS_2 = 0$). It will be inferior in terms of energy reduction in this case if b_2 has higher potential in energy reduction by claiming the time slack than b_1. Therefore, we can let the job consisting of b_1 and b_4 start earlier so that b_2 can be eligible to slowdown without affecting any other block's deadline. Essentially, we need to move part of b_4's execution with a length of *s* before b_2's arrival time. In this example, note that moving up b_3 does not help as b_3 itself arrives later than b_2. But b_3 also benefits from task rescheduling: it now can reclaim full amount of the slack. As another example, using the previous scenario, as shown in Fig. 6, b_6 to b_{10} are now legal to reclaim 1 unit of slack by moving one unit of b_7 before b_6. In general, by making the suggested changes in the schedule, *MaxRS* values of blocks in the exploration window will be larger than the case when no task rescheduling is applied. Effectively, it is equal to judiciously changing the priority of the dynamic slack (defined in [10]) so that it can be better utilized as compared to a strategy that reclaims it as soon as possible by allocating it to the very next task [10]. Note that rescheduling is attempted for deciding *MaxRS* but only actually happens when the corresponding block is selected as the target of slack reclamation.

Obviously, in case of Fig. 5, the amount of slack that b_2 as well as b_3 can reclaim depends on how much of b_4 can be brought forward. It is possible that b_1 itself has smaller *MaxRS* than what is available due to its own arrival time constraint. Besides, if the length of b_4 (under its current voltage assignment), say t_4^k, before rescheduling is shorter than the slack *s*, b_2 can only accept a slack with length of t_4^k after rescheduling. However, on the other hand, it is also possible

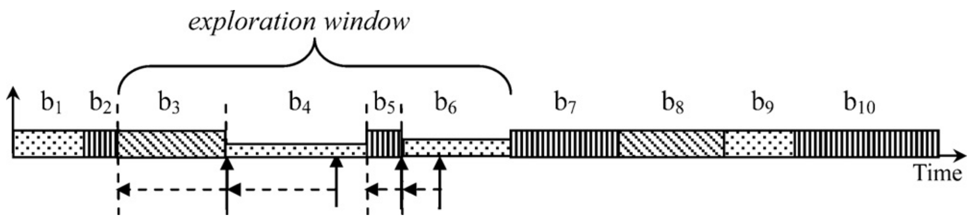


Fig. 3. Dynamic slack allocation example.

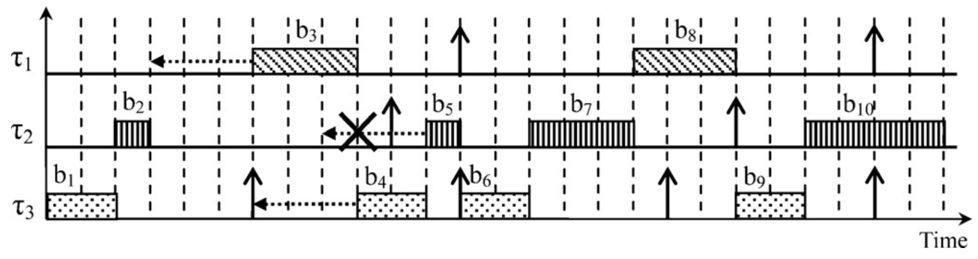


Fig. 4. Dynamic slack allocation with arrival time constraints.

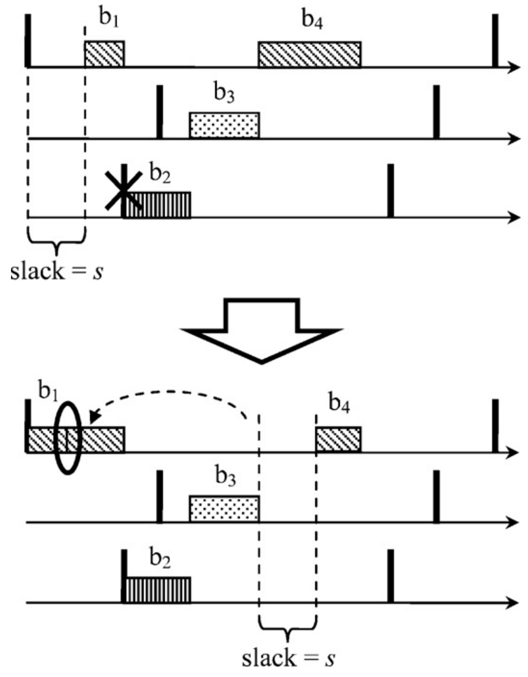


Fig. 5. Slack reclamation with task rescheduling.

that b_2 and b_3 themselves are not able to slowdown by s due to their own deadline constraints.[3] These scenarios will all result in reduced *MaxRS* for subsequent blocks inevitably. In general, there may be multiple candidates that can be moved forward for maintaining higher $MaxRS_i$. We can simply choose the one which would result in maximum value of $MaxRS_i$. Note that if b_1 and b_4 have different voltage assignments before rescheduling, an extra scaling is needed which may lead to certain amount of overhead (for setting voltage level) and need to be taken into account during decision making. It is also possible that, for some block b_i, no subsequent block of it has earlier arrival time. In this case, there is no remedy and b_i as well as all subsequent blocks can only receive a reduced *MaxRS*.

[3] Our study shows that it is rare and only happens when there are very few tasks (e.g., 2).

4.3. Slack reclamation algorithm

In this section, we describe the details of our algorithm. For tasks without arrival time constraints, all blocks in the exploration window share the same *MaxRS* which is equal to the total amount of slack. However, for tasks with arrival time, there will be a series of n' groups with all blocks in each group having equal *MaxRS* and the groups' *MaxRS* values are in decreasing order as shown in Fig. 7. This is because *MaxRS* remains the same for each block, but may monotonically decrease for consecutive blocks due to additional constraints discussed above. Therefore, we have $MaxRS_1 > MaxRS_2 > \cdots > MaxRS_{n'}$.

We define the minimum amount of slack time that can allow block b_i to lower down its current voltage level to the next available lower level as *minimum reclaimable slack* ($MinRS_i$). Note that a slack smaller than $MinRS_i$ will have no benefit for b_i since no energy saving can be achieved. If the block is already in the lowest voltage level of its profile table (thus further slow down will drop below its critical speed), its *MinRS* is set to ∞. This process is applied iteratively for the available slack. A greedy approach is used in which the energy saving per unit of slack (*ESpU*) is maximized in each iteration. Specifically, for block b_i, we have:

$$ESpU_i = \frac{e_i^{h_i} - e_i^{h_i+1}}{MinRS_i} \tag{8}$$

where h_i is the index of the current voltage level of b_i and $MinRS_i = t_i^{h_i+1} - t_i^{h_i}$. We assign *MinRS* units of slack to the block which has the maximum *ESpU* value, but has $MinRS \leqslant MaxRS$. After each iteration, the target block's *MinRS* is recalculated and each group's *MaxRS* needs to be updated sequentially in a cascading fashion. Specifically, if b_i in group i' is allocated $MinRS_i$ units of slack, we let $MaxRS_j = MaxRS_j - MinRS_i$ for all blocks b_j in group i' (including b_i) as well as all the groups before i' along the timeline. If the blocks in group i' still have their common *MaxRS* larger than the ones in the next group, no update is required for all the subsequent groups. If the *MaxRS* value for group i' drops below its next group i'+1, we have to make them equal. Since group i'+1's *MaxRS* also gets changed, the update process repeats until it reaches the last group or the next group has lower *MaxRS*.

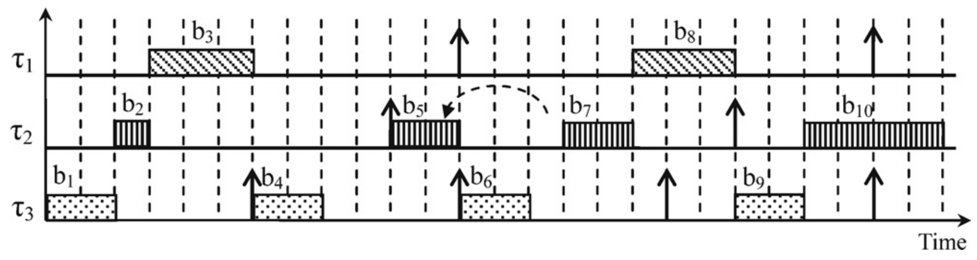


Fig. 6. Another example of slack reclamation with task rescheduling.

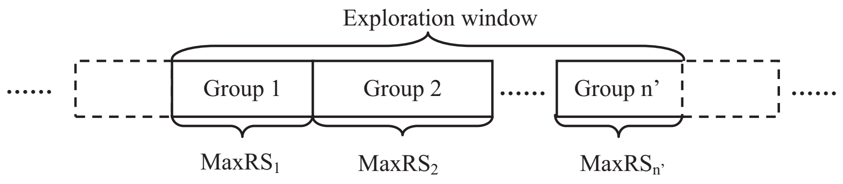


Fig. 7. Exploration window partitions into groups according to *MaxRS*.

Algorithm 1. Dynamic slack reclamation algorithm.

```
1:  Input: $startIdx$, $s$, $w$.
2:  Output: New scheduling for subsequent blocks.
3:  Step 1: Calculate $MaxRS$ for all the blocks in the window as
    discussed in Section 4.2.2.
4:  Step 2: Dynamic slack reclamation.
5:  $endIdx \leftarrow startIdx + w$;
6:  $minMinRS \leftarrow min(MinRS_i), \forall i \in [startIdx, endIdx]$;
7:  Calculate $MinRS_i$ and $ESpU_i$, $\forall i \in [startIdx, endIdx]$;
8:  while $s \geqslant minMinRS$ do
9:    Find $b_j$ in the window with:
10:   1) $MinRS_j \leqslant s$;
11:   2) $MinRS_j \leqslant MaxRS_j$;
12:   3) $ESpU_j$ is the maximum for $\forall j \in [startIdx, endIdx]$;
13:   Allocate $MinRS_j$ units of slack to $b_j$ and apply task rescheduling if
    needed as discussed in Section 4.2.2;
14:   $s \leftarrow s - MinRS_j$;
15:   Update $MinRS_j$, $ESpU_j$, $minMinRS$ and $MaxRS$ for all the blocks in
    the window as discussed in Sections 4.2.2 and 4.3;
16: end while
```

Algorithm 1 shows the outline of our approach. Let *startIdx* denote the index of the early finished block that creates slack with duration *s*. Here *w* represents the exploration window size. Note that lines 13 and 15 are done based on the problem requirements (with/without arrival time constraints, allow/deny task rescheduling) accordingly. If multiple slack pieces are created due to one early-finished job, Algorithm 1 is called separately in a reverse order starting from the latest slack with the same size of exploration window. In this case, we use a simple scheme that all the blocks in the examined windows are procrastinated by the residual amount of slack if possible. By doing that, the unused slacks tend to combine together to form a larger idle period. For single piece slack reclamation, our algorithm inherently maintains all unused slack before all the subsequent blocks. Since our approach considers multiple candidates for slack allocation, the residual slack is normally very small (i.e. the system utilization *U* is close to 1). Earlier work has shown that static procrastination has no benefit [23] and dynamic procrastination can at most improve the total energy efficiency by 1% when *U* is larger than 60% [10]. Therefore, our scheme that does not consider procrastination during scheduling will only lead to negligible solution quality degradation since there is no need to apply dynamic slack reclamation when *U* is smaller than 50%. This is because static scheduling already makes each task operating at or near the critical speed. We only consider the scenarios where *U* is no smaller than 0.6, which are reasonable and practical cases. Algorithm 1 scans all the blocks in the window once per slack allocation until the available dynamic slack is exhausted (less than *minMinRS*, precisely). Therefore the time complexity is $O(s \cdot w)$.

Table 2
Task sets consisting of real benchmarks.

Sets	Tasks
Set 1	cjpeg, pegwit, untoast, epic, mpeg2
Set 2	A2TIME01, BaseFP01, BITMNP01, RSPEED01, TBLOOK01
Set 3	susan, dijkstra, rijndael, qsort, stringsearch
Set 4	cjpeg, pegwit, A2TIME01, RSPEED01, pktflow, dijkstra

5. Experiments

5.1. Experimental setup

We evaluate our algorithm through simulation using two DVS-capable processors: Marvell StrongARM processor [2] and Transmeta Crusoe processor [3]. The former one supports four voltage–frequency levels (1.5 V–206 MHz, 1.4 V–192 MHz, 1.2 V –162 MHz and 1.1 V–133 MHz) and its characteristics are collected from manufacturer's datasheets. The latter one has scalable voltage level from 1.1 V to 1.5 V in steps of 0.1 V. Its operating frequency and power consumption values are calculated by the detailed energy model described in Section 3.2.

Synthetic tasks: We consider seven randomly generated synthetic task sets. Each set consists of 3–10 tasks. The workload of each task under the highest voltage level and the period (for periodic tasks) or inter-arrival time (for non-periodic tasks) are randomly chosen within pre-determined ranges so that at any moment *U* is maintained under the schedulability constraint (e.g. 1.00 for EDF). For each task set, we vary *U* from 0.6 to 0.9 in steps of 0.1. We define λ as the probability for a job to finish earlier than its WCET. If a job completes earlier, its ACET is generated using a normal distribution with a mean of (BCET + WCET)/2 and a standard deviation of (WCET − BCET)/6. BCET for each task is based on a percentage of its WCET and is varied from 10% to 100% in steps of 10%. Let δ denote the value of BCET/WCET.

Real benchmarks: We also evaluate our approach using real benchmarks selected from MediaBench [32], MiBench [33] and EEMBC [34] to form four task sets as shown in Table 2. Task Set 1 consists of tasks from MediaBench, Set 2 from EEMBC, Set 3 from MiBench and Set 4 is a mixture of all three suites. In Set 4, the two benchmarks from EEMBC are set to iterate 100 times in order to make their size comparable with others. We use SimpleScalar [35]

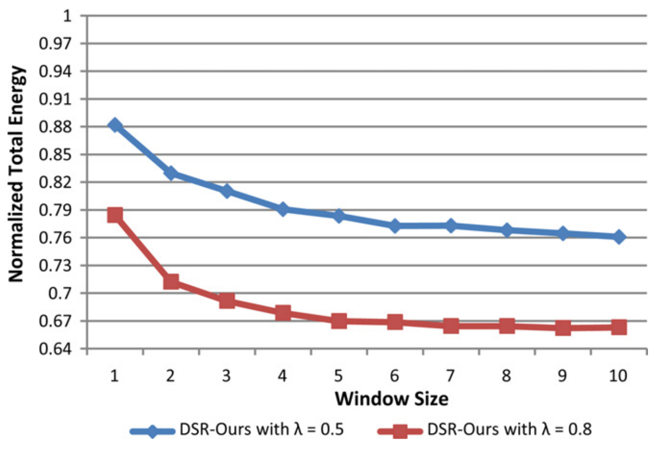


Fig. 8. Effect of Window size on the total energy savings.

as the underlying micro-architectural simulator to get the number of cycles for each task execution to act as its WCET. Task sets are formed with similar characteristics (e.g., period, phase, deadline, utilization, etc.) as synthetic task sets.

We use a static slack allocation algorithm adapted from [26]. Voltage/frequency assignment for task τ_i is the one with minimum energy consumption but has execution time no longer than $min(t_i^h/U, t_i^{crit})$, where t_i^h and t_i^{crit} is the execution time under the highest frequency level and the critical speed, respectively. Our proposed dynamic slack reclamation algorithm is implemented with a discrete-event simulator written in C++. Although our experimental results have been conducted for a large number of parameter values, we only present representative results due to space limitations.

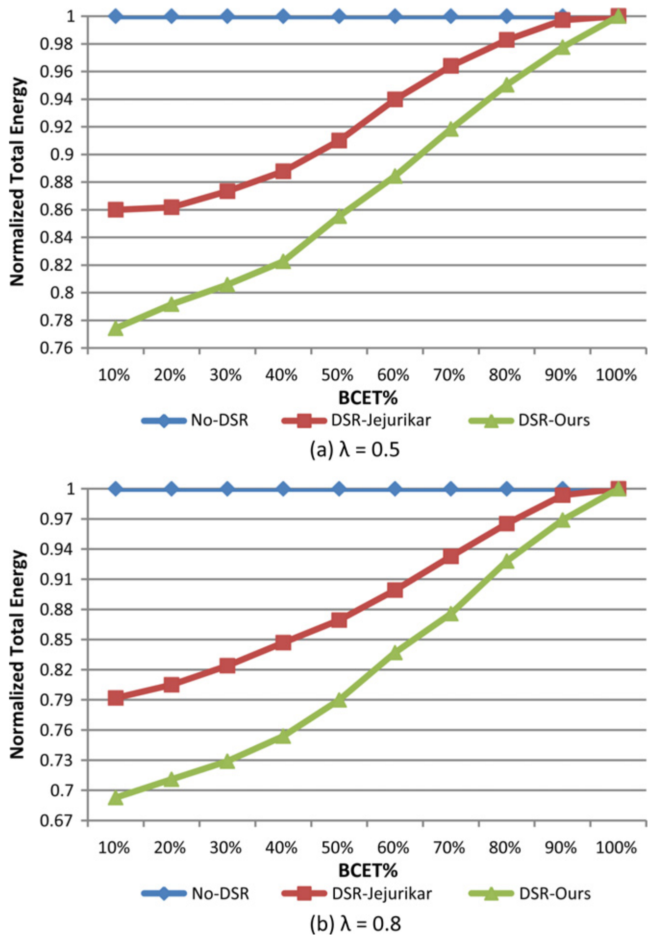


Fig. 9. Results for StrongARM processor (synthetic task sets).

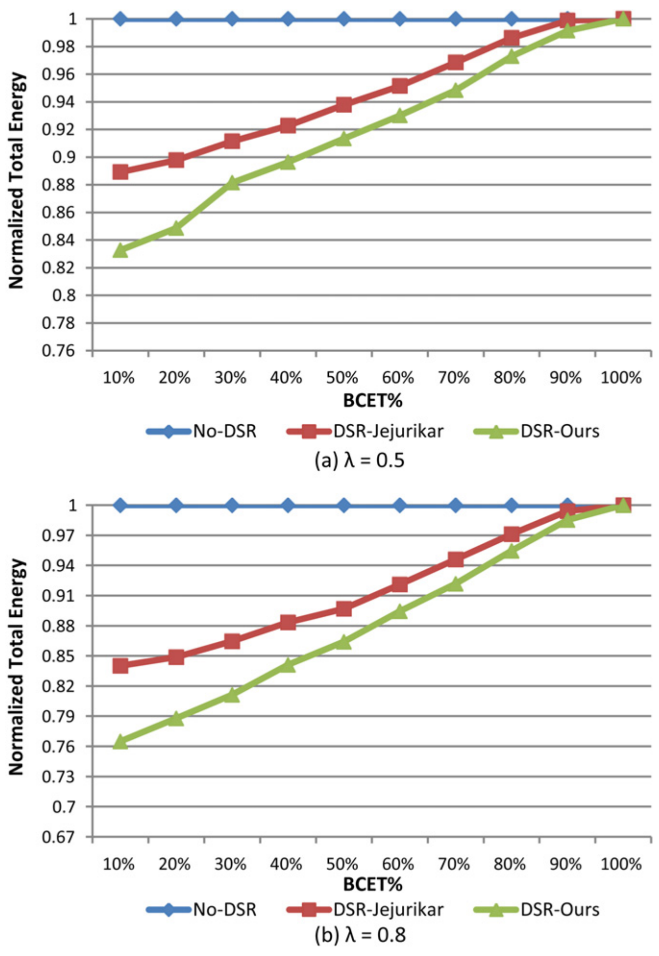


Fig. 10. Results for Transmeta Crusoe processor with constant effective capacitance values (synthetic task sets).

5.2. *Results: window size effect*

We first show the effect of adjusting the exploration window size. Here, the window size is varied from 1 to 10 with $U = 0.8$ and $\delta = 20\%$. Fig. 8 shows the average results over all synthetic task sets assuming no arrival time constraints on StrongARM processor. It shows that window size of 4 or 5 is good enough to capture most of the energy savings. Furthermore, larger window size also lead to more overhead and a higher chance that some blocks that are allocated slacks finish earlier than expected. This can compromise the total energy saving achieved. Therefore, we use window size of 4 in the following experiments.

5.3. *Results: energy saving comparison*

To illustrate effectiveness of our approach, we compare the following three techniques across different values of U and δ as discussed above:

- *No-DSR*: tasks are executed based on the static scheduling and no dynamic slack reclamation is utilized.
- *DSR-Jejurikar*: state-of-the-art dynamic slack reclamation algorithm proposed in [10].
- *DSR-Ours*: our approach on dynamic slack reclamation.

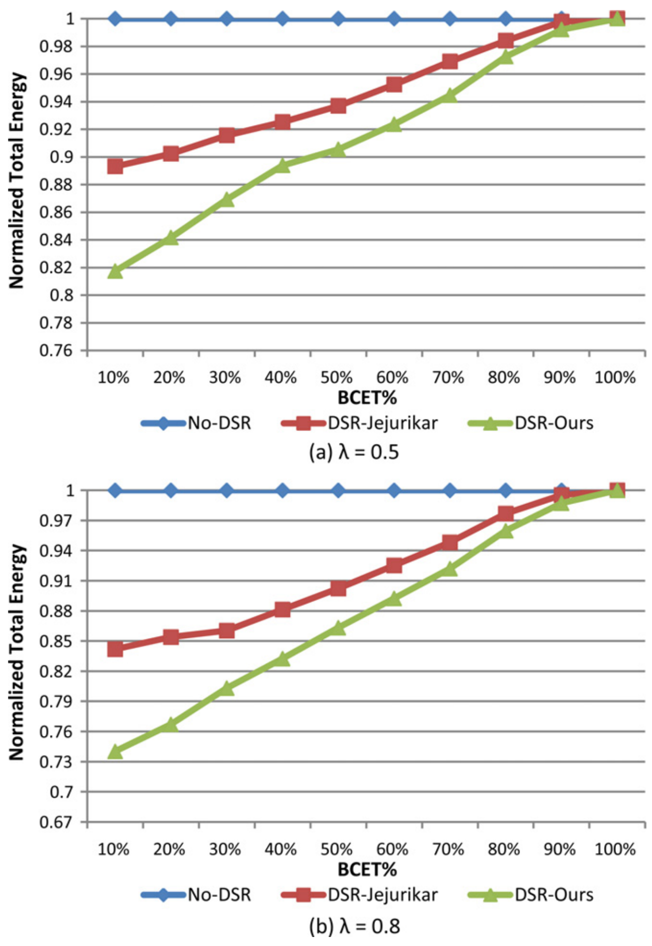


Fig. 11. Results for Transmeta Crusoe processor with application-specific effective capacitance values (synthetic task sets).

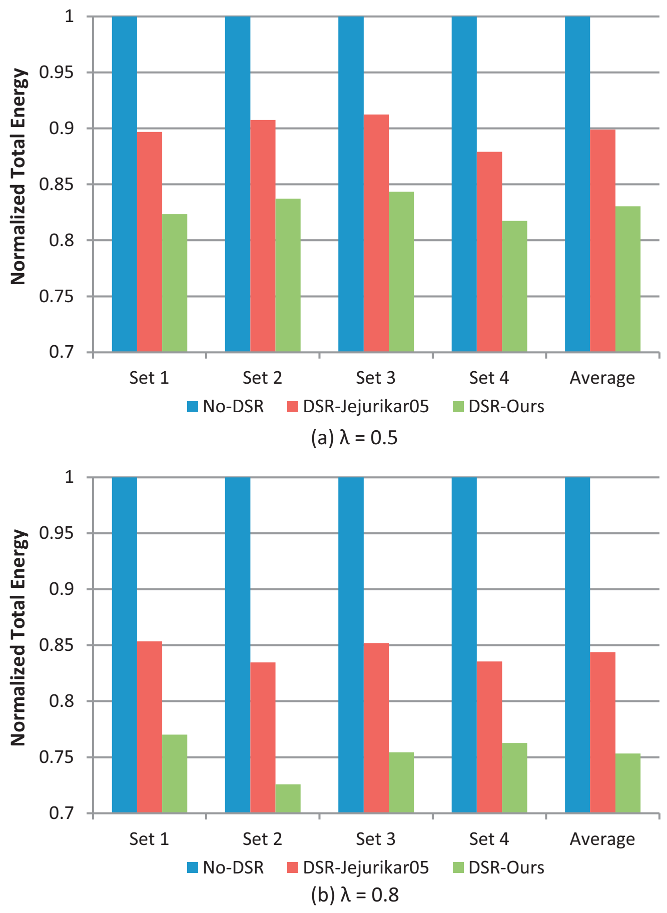


Fig. 12. Results for Transmeta Crusoe processor with application-specific effective capacitance values (real benchmark task sets).

5.3.1. *Synthetic task sets*

Fig. 9 shows energy comparison results using synthetic task sets on StrongARM processor. We examine both scenarios of (a) $\lambda = 0.5$ and (b) $\lambda = 0.8$ with window size of 4. The total energy consumption values for all techniques are normalized to No-DSR scenario. These have been averaged over multiple runs of all task sets and all values of U, where each run consists of a combination of a task set and a value of U. For both values of λ, our approach can achieve average energy savings of 14% and 18% over No-DSR (can be as large as 23% and 31% when $\delta = 10\%$). Our approach also outperforms DSR-Jejurikar across all δ values by 2–12% in terms of total energy requirements. In practice, the ACET of a program is smaller than its WCET by at least 80% (i.e. $\delta = 20\%$) [26], especially when the WCET estimation is pessimistic. In such cases, our technique can reduce the energy consumption by more than 10% compared with the state-of-the-art algorithm.

Figs. 10 and 11 show the results for the same set of experiments on Transmeta Crusoe processor with constant effective capacitance and application-specific effective capacitance, respectively. For the latter case, we randomly generate $\mathcal{K}$ in the energy model within a range of [0.2, 1.0] for each task. In other words, we have $C_{eff} \in [0.2 \cdot C_{total}, C_{total}]$. In both scenarios, it can be observed that energy savings are less significant than StrongARM processor. It is possibly due to the fact that leakage energy consumption is much higher in 70 nm technology. Therefore, the energy reduction created by DSR (lower subsequent job's voltage level) decreases. However, our approach still consistently outperforms DSR-Jejurikar. Another important observation is that in the scenario where tasks have different effective capacitance (C_{eff}), our approach can result in more additional energy savings compared with DSR-Jejurikar. The reason is that application-specific C_{eff} leads to more variation in task's energy saving potential during dynamic slack reclamation, which clearly makes our approach more beneficial.

5.3.2. *Real benchmarks*

Fig. 12 shows total energy consumption comparisons across four real benchmark task sets with $\delta = 10\%$ and (a) $\lambda = 0.5$, (b) $\lambda = 0.8$ on Transmeta Crusoe processor. Here, similar observation can be made as shown in Fig. 11. On average, 7% and 10% extra savings in total energy consumption can be achieved in both scenarios, respectively.

5.4. *Results: problem variations*

To demonstrate the breadth of applicability of our approach, we compare the experimental results for the following three scenarios: (1) *No-AT*: task sets without arrival time constraints (Section 4.1); (2) *AT-NoRS*: tasks with arrival time constraints but task rescheduling is not allowed (Section 4.2.1); (3) *AT-RS*: tasks with arrival time constraints but task rescheduling is applied (Section 4.2.2). λ and U are set to 0.8. It can be observed from Fig. 13 that task rescheduling is very effective and can achieve energy savings very close to No-AT. Thus, our approach is able to exploit the available slack effectively even when significant constraints on task rescheduling and arrival times are considered.

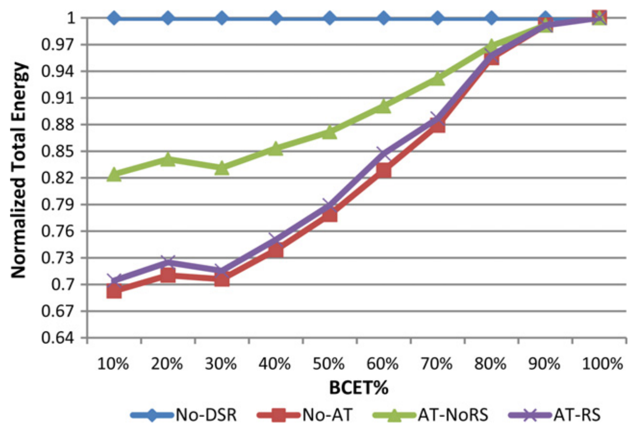


Fig. 13. Problem variations comparison.

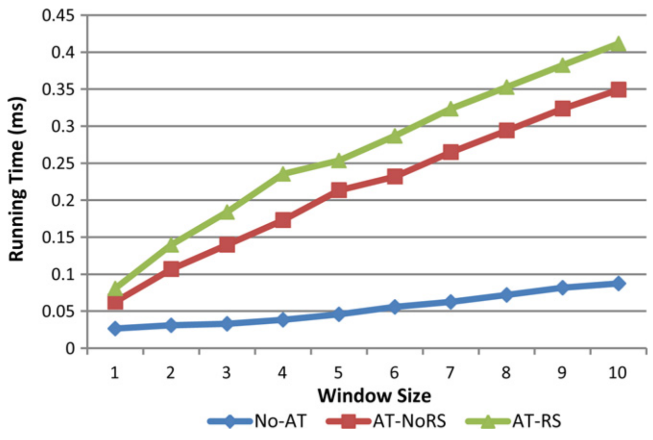


Fig. 14. Running time overhead.

5.5. *Results: running time overhead*

We also investigated the runtime overhead of our DSR algorithm for all three scenarios above. Fig. 14 shows the average running time requirement (of one time of dynamic slack reclamation) over all task sets with λ and δ equal to 0.8 and 0.2, respectively. The window size is varied from 1 to 10. We can observe that the running time overhead of AT-RS is very low (e.g. less than one fourth millisecond for window size of 4). Therefore, our algorithm is efficient enough at runtime for normal task sets which normally takes hundreds of milliseconds [36,37].

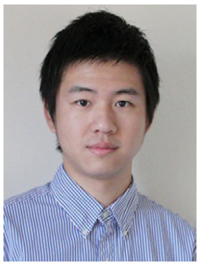
6. Conclusion

In this paper, we presented a dynamic slack reclamation algorithm for energy-aware scheduling in real-time multitasking systems. Our approach aims at minimizing total energy consumption, both dynamic and leakage, when some tasks finish earlier than their worst case. Unlike existing techniques, we systematically allocate the available slack among multiple jobs and apply task rescheduling whenever it is beneficial. By restricting the exploration window, tradeoffs can be made between solution quality and runtime overhead. Experimental results show that our approach can achieve significant energy saving over static energy-aware scheduling and also outperforms state-of-the-art technique by up to 12%.

References

[1] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, M.B. Srivastava, Power optimization of variable-voltage core-based systems, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18 (1999) 1702–1714.
[2] Marvell, StrongARM 1100 processor, 1997. http://www.marvell.com/.
[3] Transmeta, Transmeta Crusoe Processor, 2001. http://www.transmeta.com/.
[4] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, D. Shippy, Introduction to the cell multiprocessor, IBM Journal of Research and Development 49 (2005) 589–604.
[5] S. Borkar, Design challenges of technology scaling, IEEE Micro 19 (4) (1999) 23–29, http://dx.doi.org/10.1109/40.782564.
[6] N.S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, V. Narayanan, Leakage current: Moore's law meets static power, Computer 36 (12) (2003) 68–75, http://dx.doi.org/10.1109/MC.2003.1250885, ISSN 0018-9162.
[7] J.A. Butts, G.S. Sohi, A static power model for architects, in: Proc. 33rd Annual IEEE/ACM International Symposium on MICRO-33 Microarchitecture, 2000, pp. 191–201, http://dx.doi.org/10.1109/MICRO.2000.898070.
[8] G. Buttazzo, Hard Real-Time Computing Systems, Kluwer, 1995.
[9] J. Liu, Real-Time Systems, Prentice Hall, 2000.
[10] R. Jejurikar, R. Gupta, Dynamic slack reclamation with procrastination scheduling in real-time embedded systems, in: Proceedings of Design Automation Conference, 2005, pp. 111–116.
[11] H. Aydin, R. Melhem, D. Mosse, P. Mejia-Alvarez, Determining optimal processor speeds for periodic real-time tasks with different power characteristics, in: 13th Euromicro Conference on Proc. Real-Time Systems, 2001, pp. 225–232, http://dx.doi.org/10.1109/EMRTS.2001.934038.
[12] F. Yao, A. Demers, S. Shenker, A scheduling model for reduced CPU energy, in: Proceedings of Annual Symposium on Foundations of Computer Science, 1995, pp. 374–382.
[13] H. Aydin, R. Melhem, D. Mosse, P. Mejia-Alvarez, Power-aware scheduling for periodic real-time tasks, IEEE Transactions on Computers 53 (5) (2004) 584–600, http://dx.doi.org/10.1109/TC.2004.1275298.
[14] D. Shin, J. Kim, Dynamic voltage scaling of periodic and aperiodic tasks in priority-driven systems, in: Design Automation Conference, 2004. Proceedings of the ASP-DAC 2004. Asia and South Pacific, 2004, pp. 653–658.
[15] X. Zhong, C. Xu, System-wide energy minimization for real-time tasks: lower bound and approximation, in: Proceedings of International Conference on Computer-Aided Design, 2006, pp. 516–521.
[16] R. Jejurikar, R. Gupta, Energy aware non-preemptive scheduling for hard real-time systems, in: Proc. 17th Euromicro Conference on Real-Time Systems (ECRTS 2005), 2005, pp. 21–30, http://dx.doi.org/10.1109/ECRTS.2005.13.
[17] J. Chen, T. Kuo, C. Shih, 1 +ε approximation clock rate assignment for periodic real-time tasks on a voltage-scaling processor, in: Proceedings of International Conference on Embedded Software, 2005, pp. 247–250.
[18] S. Zhang, K. Chatha, G. Konjevod, Approximation algorithms for power minimization of earliest deadline first and rate monotonic schedules, in: Proceedings of International Symposium on Low Power Electronics and Design, 2007, pp. 225–230.
[19] D. Shin, J. Kim, Optimizing intratask voltage scheduling using profile and data-flow information, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 26 (2007) 369–385.
[20] S. Oh, J. Kim, S. Kim, C. Kyung, Task partitioning algorithm for intra-task dynamic voltage scaling, in: Proceedings of International Symposium on Circuits and Systems, 2008, pp. 1228–1231.
[21] W. Wang, P. Mishra, PreDVS: preemptive dynamic voltage scaling for real-time systems using approximation scheme, in: Proceedings of Design Automation Conference, 2010, pp. 705–710.
[22] Y.-H. Lee, K. Reddy, C. Krishna, Scheduling techniques for reducing leakage power in hard real-time systems, in: Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on, 2003, pp. 105–112.
[23] R. Jejurikar, C. Pereira, R.K. Gupta, Leakage aware dynamic voltage scaling for real-time embedded systems, in: Proceedings of Design Automation Conference, 2004, pp. 275–280.
[24] J.-J. Chen, T.-W. Kuo, Procrastination for leakage-aware rate-monotonic scheduling on a dynamic voltage scaling processor, in: LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems, ACM, New York, NY, USA, 2006, http://dx.doi.org/10.1145/1134650.1134673, ISBN 1-59593-362-X, 153-162.
[25] J.-J. Chen, T.-W. Kuo, Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems., in: Proc. IEEE/ACM International Conference on Computer-Aided Design ICCAD 2007, 2007, pp. 289–294, http://dx.doi.org/10.1109/ICCAD.2007.4397279.
[26] H. Aydin, R. Melhem, D. Mosse, P. Mejia-Alvarez, Dynamic and aggressive scheduling techniques for power-aware real-time systems, in: Proceedings of Real-Time Systems Symposium, 2001, pp. 95–105.
[27] P. Pillai, K. Shin, Real-time dynamic voltage scaling for low-power embedded operating systems, ACM SIGOPS Operating Systems Review (2001) 89–102.
[28] W. Kim, J. Kim, S. Min, A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis, in: Proceedings of Design, Automation and Test Conference in Europe, 2002, p. 788.
[29] W. Wang, P. Mishra, Leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in real-time systems, in: Proceedings of IEEE International Conference on VLSI Design, 2010, pp. 357–362.

[30] S.M. Martin, K. Flautner, T. Mudge, D. Blaauw, Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads, in: Proc. IEEE/ACM International Conference on Computer Aided Design ICCAD 2002, 2002, pp. 721–725, http://dx.doi.org/10.1109/ICCAD.2002.1167611.
[31] W. Wang, P. Mishra, System-wide leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in multitasking systems, IEEE Transactions on Very Large Scale Integration Systems (TVLSI) (2011) 1–9.
[32] C. Lee, M. Potkonjak, W.H. Mangione-smith, MediaBench: a tool for evaluating and synthesizing multimedia and communications systems, in: Proceedings of International Symposium on Microarchitecture, 1997, pp. 330–335.
[33] M. Guthaus, J. Ringenberg, D. Ernest, T. Austin, T. Mudge, R. Brown, MiBench: a free, commercially representative embedded benchmark suite, in: Proceedings of IEEE International Workshop on Workload Characterization, 2001, pp. 3–14.
[34] EEMBC, The Embedded Microprocessor Benchmark Consortium, EEMBC, 2000.
[35] D. Burger, T.M. Austin, S. Bennett, Evaluating future microprocessors: the SimpleScalar tool set, Tech. Rep., University of Wisconsin-Madison, 1996.
[36] S. Zhang, K.S. Chatha, Approximation algorithm for the temperature aware scheduling problem, in: Proceedings of International Conference on Computer-Aided Design, 2007, pp. 281–288.
[37] W. Wang, X. Qin, P. Mishra, Temperature- and energy-constrained scheduling in multitasking systems: a model checking approach, in: Proceedings of International Symposium on Low Power Electronics and Design, 2010, pp. 85–90.



Weixun Wang received his B.E. degree in software engineering from the Software Institute, Nanjing University, Nanjing, China, in 2007. He is currently pursuing his Ph.D. degree in the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, USA. His research interests include the area of design automation of embedded systems with focus on dynamic cache reconfiguration, energy optimization, temperature management, design space exploration and lossless data compression.

Sanjay Ranka is a Professor in the Department of Computer Information Science and Engineering at University of Florida. His current research interests are energy efficient computing, high performance computing, data mining and informatics. Most recently he was the Chief Technology Officer at Paramark where he developed real-time optimization software for optimizing marketing campaigns. Sanjay has also held positions as a tenured faculty positions at Syracuse University and as a researcher/visitor at IBM T.J. Watson Research Labs and Hitachi America Limited.

He earned his Ph.D. (Computer Science) from the University of Minnesota and a B. Tech. in Computer Science from IIT, Kanpur, India. He has coauthored two books: Elements of Neural Networks (MIT Press) and Hypercube Algorithms (Springer Verlag), 70 journal articles and 110 refereed conference articles. His recent work has received a student best paper award at ACM-BCB 2010, best paper runner up award at KDD-2009, a nomination for the Robbins Prize for the best paper in journal of Physics in Medicine and Biology for 2008, and a best paper award at ICN 2007.

He is a fellow of the IEEE and AAAS, and a member of IFIP Committee on System Modeling and Optimization. He serves on the editorial board of Journal of Parallel and Distributed Computing, IEEE Transactions on Parallel and Distributed Computing, Sustainable Computing: Systems and Informatics, and International Journal of Computing.

He was a past member of the Parallel Compiler Runtime Consortium, the Message Passing Initiative Standards Committee and Technical Committee on Parallel Processing. He is the program chair for 2010 International Conference on Contemporary Computing and co-general chair for 2009 International Conference on Data Mining and 2010 International Conference on Green Computing.

Prabhat Mishra is an Associate Professor in the Department of Computer and Information Science and Engineering (CISE) at the University of Florida where he leads the CISE Embedded Systems Group. His research interests include design automation of embedded systems, energy-aware computing, hardware/software verification, and design of trustworthy systems. He received his B.E. from Jadavpur University, Kolkata in 1994, M.Tech. from the Indian Institute of Technology, Kharagpur in 1996, and Ph.D. from the University of California, Irvine in 2004 – all in Computer Science. Prior to joining University of Florida, he spent several years in various semiconductor and design automation companies including Intel, Motorola, Synopsys and Texas Instruments. He has published four books, ten book chapters and more than 100 research articles in premier international journals and conferences. His research has been recognized by several awards including the NSF CAREER Award from the National Science Foundation, two best paper awards (VLSI Design 2011 and CODES + ISSS 2003), and 2004 EDAA Outstanding Dissertation Award from the European Design Automation Association. Prof. Mishra currently serves as an Associate Editor of ACM Transactions on Design Automation of Electronic Systems, IEEE Design & Test of Computers, and Springer Journal of Electronic Testing, Guest Editor of IEEE Transactions on Computers, and as a program/organizing committee member of several premier ACM and IEEE conferences. He is a senior member of both ACM and IEEE.