A Practical Slack-time Analysis Method for DVS Real-time Scheduling

Da-Ren Chen, You-Shyang Chen, Min-Fong Lai

A Practical Slack-time Analysis Method for DVS Real-time Scheduling

Da-Ren Chen
Department of information Management,
Hwa Hsia Institute of Technology,
Taipei, Taiwan.
danny@cc.hwh.edu.tw

You-Shyang Chen
Department of information Management,
Hwa Hsia Institute of Technology,
Taipei, Taiwan.
ys_chen@cc.hwh.edu.tw

Min-Fong Lai
Science and Technology Policy Research and Information Center,
National Applied Research Laboratories, Taipei, Taiwan, R.O.C.
e-mail:danny@cc.hwh.edu.tw

***Abstract*—This work presents a scheduling algorithm to reduce the energy of hard real-time tasks with fixed priorities assigned in a rate-monotonic policy. Sets of independent tasks running periodically on a processor with dynamic voltage scaling (DVS) are considered as well. The proposed online approach can cooperate with many slack-time analysis methods based on low-power work demand analysis (lpWDA) without increasing the computational complexity of DVS algorithms. The proposed approach introduces a novel technique called low-power fluid slack analysis (lpFSA) that extends the analysis interval produced by its cooperative methods and computes the available slack in the extended interval. The lpFSA regards the additional slack as *fluid* and computes its length, such that it can be moved to the current job. Therefore, the proposed approach provides cooperative methods with additional slack. Experimental results show that the proposed approach combined with lpWDA-based algorithms achieves more energy reductions than do the initial algorithms alone.**

Keywords- real-time systems, fixed-priority scheduling, dynamic voltage scaling, slack time analysis

1. INTRODUCTION

Dynamic voltage scaling (DVS) is a standard technique for managing the power consumption of the systems [23]. A DVS processor can vary its operation frequency and voltage during its runtime to use the quadratic relationship between energy consumption and supply voltage of CMOS technology. In the recent years, computation and communication have been steadily moved toward mobile and portable devices with limited power supply. Therefore, many primary IC producers have developed their modern processors with DVS capability, including Intel's XScale®[10], AMD's mobile Athlon®[1] and SamSung's Cortex® [21].

There are many substantial researches for scheduling real-time applications on DVS processors [2, 5, 6, 8, 9, 11, 19, 20, 22]. These approaches differ in many aspects, such as the scheduling algorithms being on-line/off-line, handling discrete/continuous voltage levels, assuming average-case, best-case and worst-case execution times (ACET, BCET and WCET) of each task, allowing intra-task/inter-task voltage transitions and assuming fixed/dynamic priority assignment. However, they still have common objective and meet the same difficulty. Because lowering the supply voltage also decreases the maximum achievable clock speed [16], most DVS algorithms for real-time systems try to reduce the supply voltage dynamically to the lowest possible speed level while satisfying the systems' soft/hard timing constraints. In order to satisfy the timing constraints of the real-time tasks, dynamic voltage scaling can utilize slack times when adjusting voltage levels. Consequently, the energy efficiency of a DVS algorithm highly depends on the accuracy of estimating the length of slack.

Many previous researches have been conducted regarding slack time analysis [5, 8, 11, 12, 13, 17, 19]. Lehoczky et al. proposed a slack stealing algorithm [13] which creates a passive task called *slack stealer*. It attempts to make time for servicing aperiodic tasks by stealing all the processing time it can from the periodic tasks without deadline missing. The slack stealer relies on the schedulability conditions given by Lehoczky et al. [14] and Lehoczky [15] to provide the maximum possible capacity for aperiodic service at the time it is required. In addition, they proposed an extension of the algorithm called *reclaimer*. It utilizes the pre-allocated but unused worst-case execution time (WCET) to improve system performance. Lorch et al. [17] proposed a probabilistic method called Processor Acceleration to Conserve Energy (PACE) to keep system performance while minimize expected energy consumption. Since PACE depends on the probability distribution of the task's work load, it must estimates beforehand the distribution of task work from the requirements of previous, similar tasks. Pillai and Shin [19] proposed a cycle-conserving rate-monotonic (ccRM) scheduling, which contains off-line and on-line algorithms. The off-line algorithm computes the worst-case response time of each task and derives the maximum speed that need to meet all task deadlines. When a task instance completes early, the on-line algorithm proceed to scale down the processor speed. Algorithm ccRM is a conservative method because it only considers the possible slack time before the next task arrival (NTA) of current job. In [8], Gruian proposed the methods of off-line task stretching and on-line slack distribution. Gruian's off-line method is also conservative, and its on-line slack time analysis is based on a probability distribution function. Kim et al. [11] proposed a greedy on-line algorithm called low-power work-demand analysis (lpWDA) that derives slack from lower priority tasks, as opposed to the method in [8, 19] that gain slack time

from higher priority tasks. It also balances the gap of voltage levels between higher and lower priority tasks. They have shown that lpWDA always produces a valid schedule as long as transition time overhead is negligible. However lpWDA is still a conservative method because its slack time analysis is confined in an analysis scope regarding to the longest length of task periods. There are also many slack time analysis methods considering additional assumptions [5, 9, 12, 18]. In [12], Kim et al. proposed a preemption-aware DVS algorithm based on lpWDA, which is composed of *accelerated-completion* and *delayed-preemption* techniques (lpWDA-AC and lpWDA-DP, respectively) to decrease the preemption times of DVS algorithms. The lpWDA-AC tries to avoid preemption by adjusting the voltage/clock speed higher than the lowest possible values computed using lpWDA. Another technique, lpWDA-DP, postpones preemption points by delaying an activated higher-priority task as late as possible while guaranteeing the feasible schedule of tasks. Both techniques can reduce more energy consumption when compared to initial ccRM and lpWDA. Mochocki et al. in [118] also proposed a transition-aware DVS algorithm for decreasing the number of voltage/speed adjusting, called low power limited demand analysis with transition overhead (lpLDAT), accounts for both time and energy transition overheads. It computes an efficient speed level based on the average-case workload, this speed can be used as a *limiter*. If the *limiter* is higher than the speed predicted by lpWDA, lpLDAT knows that lpWDA is being too aggressive and applies the limiter in the present schedule. This average-case limiter technique with slack time analysis also contributes much energy saving when compared to previous methods. He et al. [9] considered a fixed-priority scheduling with threshold (FPPT) which eliminates unnecessary context switches, thereby saving energy. FPPT gives each task with a pair of predefined priority and corresponding preemption threshold. They proposed an algorithm to compute the static slowdown factors by formulating the problem as a linear optimization problem. In addition, they considered the energy consumption of task set under different preemption threshold assignments. Chen and Hsu [5] proposed a tree structure corresponding to a set of pinwheel tasks [16]. They also proposed a DVS algorithm called lpJCRT which manipulates a tree structure to distribute available slack evenly among other tasks.

In this paper, we focus on enhancing the on-line slack computation capability of RM DVS algorithms. Based on the existing RM DVS algorithms we propose a dynamic slack-time computation scheme using a *slack fluid analysis* which computes the length of potential slack in the interval that is longer than the longest task period. Fluid slack analysis can be applied to many up-to-date RM DVS scheduling with various assumption including transition and preemption restrictions.

This work improves the on-line slack computation capability of RM DVS algorithms. Based on existing RM DVS algorithms, we propose an on-line slack-time computation scheme using fluid slack analysis, which computes the length of potential slack in an interval longer than the longest of task periods. The proposed method does not need to compute or perform a simulation for stochastic data, which varies according to different applications. With a slight modification1, lpFSA can be applied to many RM DVS scheduling scheme with various assumptions, including transition and preemption criteria. Our method does not increase the time complexity of given algorithms having O(n) time complexity where n denotes the number of tasks. Experimental results show that most existing RM DVS algorithms equipped with our method can reduce the energy consumption by 11% to 25% over the previous algorithms.

The rest of this paper is organized as follows. In Section2, we explain the motivation of this work. The basic idea of fluid slack analysis is proposed in Section 3. We also describe the details of the technique and algorithm in Section 4. We present the performance evaluation in Section 5 and conclude with the summary and future work in Section 6.

2. PRELIMINARIES

In this section, we present the assumptions of the task systems and introduce the necessary notations. This paper focuses on how to gain additional slack for the up-to-date RM DVS scheduling scheme. Many slack-time analysis techniques with different purposes (e.g., transition-aware, preemption-aware, etc.) can utilize lpFSA easily, they are called as the *host* algorithms of lpFSA. We outline the idea of lpWDA and lpLDA algorithms in this paper, other techniques such as lpWDA-AC, lpWDA-DP [12], and lpLDAT [18] are abridged.

2.1. System Model

We consider the preemptive hard real-time systems in which periodic real-time tasks are scheduled under the RM scheduling policy. The DVS processor used in the model operates at a finite set of supply voltage levels $V=\{v_1, \ldots, v_{\max}\}$, each with an associated speed. We normalize the processor speed by $S_{\max}$ corresponding to $v_{\max}=1$, giving $S=\{s_1,\ldots,1\}$. A set of n periodic tasks is denoted by $T=\{\tau_1,\tau_2\ldots\tau_n\}$, where the tasks are assumed to be mutually independent. Each task τ_i is described by its worst case execution cycles wc_i, and average case execution cycles ac_i ($wc_i \geqq ac_i$). Throughout this paper, the execution cycle of each task is called *work* for short. In addition, each task τ_i has a shorter period length p_i (i.e., a higher priority) than that of τ_j if $i<j$ and p_n denotes the longest task period. The relative deadline d_i of τ_i is assumed to be equal to its period p_i. Each task is invoked periodically by a *job*, and the k-th job of task τ_i is denoted as $\tau_{i,k}$. The first job of each task is assumed to be activated at time t=0. Each job is described by a release time, $r_{i,j}$, deadline, $d_{i,k}$, the number of cycles that

have already been executed ex_i^k. During run-time, we refer to the earliest job of each task that has not completed execution as the *current* job for that task, and we index that job with *cur*. The deadline of the current job for task τ_i is d_i^{cur}, and ex_i^{cur} the number of cycles that the current job of τ_i has executed.

Without loss of generality, whenever τ_i is the first scheduled task after time $r_{n,k-1}$, where $i \neq n$, the ***border*** denotes the next release time of τ_n (i.e., the $r_{n,k}$). In the fluid slack analysis method, we estimate the available slack in the interval [*border*, $r_{n,k+1}$).

2.2. Low Power Work Demand Analysis (lpWDA)

In this section, we introduce briefly an on-line DVS algorithm called lpWDA [11]. Notations e^{Exchange}, $\ell_\alpha^{\text{right}}$ and τ_{asyn} belonging to lpFSA are present in Section 4. In the line 2 of Algorithm 1, ε denotes an infinitesimal value and *readyQ* contains the currently activated tasks whose subset $\Gamma_\alpha^{ACT}(t)$ is denoted as

$$\Gamma_\alpha^{ACT}(t) := \{\, \tau_\kappa \mid \kappa < \alpha \text{ and } \tau_\kappa \in \textit{readyQ}(t) \,\}.$$

Among the tasks in *readyQ*, the active task τ_α with the shortest period is scheduled to run under the RM scheduling. When τ_α is executed at time *t*, $load_\alpha(t)$ denotes the amount of work required to be processed in $[t, d_\alpha)$. If $load_\alpha(t)$ amount of work should be completed before d_α, the slack time $slack_\alpha(t)$ is denoted as

$$d_\alpha(t)\text{-}t\text{-}load_\alpha(t).$$

Given the Algorithms 1, 2, 3 and 4 in Figure 1, lpWDA works in the following steps. First, the system is initialized by setting the initial upcoming deadlines (*ud*) and remainder execution (w^{rem}) of each task. The initial value of $H_\alpha(t)$, which denotes the estimation of the higher priority work that must be executed before ud_α (lines 1-2). The value of ε is positive and extremely small. Whenever a job τ_α is completed or preempted at time *t*, the remainder work $w_\alpha^{rem}(t)$, upcoming deadline ud_α and higher priority work $H_\alpha(t)$ are updated in line 5. In the lines 6 and 9, when a job τ_α is scheduled for execution at time *t*, Algorithm 2 computes the slack that is available for τ_α according to $H_\beta(t)$ and $L_\beta(t)$ (see lines 16 and 17), where ud_β is the earliest upcoming deadline with respect to τ_α. Notably, function $L_\beta(t)$ computing the amount of lower priority work is performed recursively until it finds τ_γ with the longest task period and lowest priority with respect to τ_α. The lpWDA computes the length of the slack-time stealing from lower priority tasks in the interval [I_α, *border*] and applying the slack to the currently executing job. Therefore, Algorithms 2 and 3 play the crucial role for slack-time analysis and dominate the run time complexity of lpWDA.

Algorithm1:lpWDA (lpFSA with ***,and lpLDA with ⋇⋇⋇)
Compute the available execution time and set the voltage/clock speed for τ_α

1. set $ud_\alpha := p_\alpha$, $e^{\texttt{Exchange}} := 0$ and $\tau_{\texttt{asyn}} := \emptyset$;
2. Compute $H_\alpha(t) := \sum_{\tau_k \in \Gamma_\alpha^{ACT}(t)} w_k^{rem}(t) + \sum_{i=1}^{\alpha-1} \left(\lfloor \frac{ud_\alpha - \varepsilon}{p_i} \rfloor - \lceil \frac{t+\varepsilon}{p_i} \rceil \right) \cdot w_i$;
3. ⋇⋇⋇$A_i := \sum_{j=0}^{i-1} \left(\lceil \frac{d_i^{cur}}{p_j} \rceil \times ac_j \right)$;
4. When a job τ_α is activated, set $w_k^{rem}(t) := w_\alpha$;
5. When a job τ_α is completed or preempted, update LoadInfo($w_\alpha^{rem}(t), ud_\alpha, H_\alpha(t)$);
6. When a job τ_α is scheduled for execution
7. ***$e^{\texttt{Exchange}}$:=**lpFSA**(t, T); //get additional slack time
8. ***if** $r_\alpha = \tau_{\texttt{asyn}}$ and $\ell_\alpha^{\texttt{right}} < e^{\texttt{Exchange}}$ **then** τ_α has the lowest priority in *readyQ*(t);
9. $slack_\alpha(t) :=$ **ClacSlackTime**$(e^{\texttt{Exchange}})$; //get slack time
10. set the clock frequency as $f_{clk} := \frac{w_\alpha^{rem}(t)}{slack_\alpha(t) + w_\alpha^{rem}(t)} \cdot f_{\max}$;
11. ⋇⋇⋇$f_{ACL} := \max\{ \frac{A_i + ac_\alpha - ex_\alpha^{cur}(t)}{d_i^{cur} - t} | i = 1, \ldots, n \}$;
12. ⋇⋇⋇$f_{clk} := \max\{ f_{clk}, f_{ACL} \}$;
13. Set the voltage accordingly;

Algorithm 2: **CalcSlackTime**(additional slack $e^{\texttt{Exchange}}$)
Input: the active task τ_α , *readyQ* and current time t
Output: the slack time $slack_\alpha(t)$ for τ_α

14. Identify the task τ_β that has the earliest upcoming deadline among tasks whose priorities are not higher than that of τ_a;
15. $L_\beta(t) :=$ **CalcLowerPriorityWork** $(\tau_\beta, t, e^{\texttt{Exchange}})$;
16. $load_\beta(t) := w_\beta^{rem}(t) + H_\beta(t) + L_\beta(t)$;
17. $slack_\alpha(t) := \max(0, ud_\beta - t - load_\beta)$;
18. **return** $(slack_\alpha(t))$;

Algorithm 3: **CalcLowerPriorityWork**(additional slack $e^{\texttt{Exchange}}$)
Input: a reference task τ_β and current time t
Output: the amount of lower-priority work needed to be done before ud_β

19. **if** τ_β is identical to τ_n **then** return 0;
    end if
20. Identify the task τ_r that has the earliest upcoming deadline among the tasks whose priorities are lower than that of τ_β;
21. $L_r(t) :=$ **CalcLowerPriorityWork** $(\tau_r, t, e^{\texttt{Exchange}})$;
22. $load_r(t) := w_\beta^{rem}(t) + H_r(t) + L_r(t)$;
23. ****if** ud_r is multiple of p_n **then** $ud_r := ud_r + e^{\texttt{Exchange}}$;
24. $L_\beta(t) :=$max$(0, load_r(t) - w_\beta(t) - H_\beta(t) - ud_r + ud_\beta)$;
25. **return** $L_\beta(t)$;

Algorithm 4: **UpdateLoadInfo**$(w_\alpha^{rem}, ud_\alpha, H_\alpha(t))$
Input: The completed or preempted task τ_α and the amount of work w_{done} done for τ_α in the previous schedule
Output: Workloads are update to reflect current execution information

26. **if**(COMPLETION)**then**
27. $ud_\alpha := ud_\alpha + p_\alpha$;
28. $H_\alpha(t) := \sum_{i=1}^{\alpha-1} \left(\lfloor \frac{ud_\alpha - \varepsilon}{p_i} \rfloor - \lceil \frac{t+\varepsilon}{p_i} \rceil \right) \cdot w_i$;
29. **loop** from $k := \alpha - 1$ **until** $k = n$ by increasing k
30. $H_l(t) := H_k(t) - w_\alpha^{rem}(t)$;
31. ⋇⋇⋇$A_k := A_k -$ max $\{0, ac_\alpha, ex_\alpha^{cur}\}$;
32. **loop** from $k := 1$ **until** $k < \alpha$ by increasing k
33. ⋇⋇⋇$A_\alpha := A_\alpha + \sum_{j=1}^{\alpha-1} \left(\lceil \frac{d_i^{cur}}{p_j} \rceil - \lfloor \frac{d_i^{cur} - p_\alpha}{p_j} \rfloor - \right) \times ac_j$;
34. esle (PREEMPTION)
35. $temp := w_\alpha - w_{done}$;
36. **loop** from $k := \alpha - 1$ **until** $k = n$ by increasing k
37. $H_k(t) := H_k(t) - w_\alpha^{rem}(t) + temp$;
38. ⋇⋇⋇$A_k := A_k - w_\alpha^{rem}(t) + temp$;
39. $w_\alpha^{rem}(t) := temp$;
40. **end if**

Figure 1. lpWDA algorithm.

2.3. Low Power Limit Demand Analysis (lpLDA)

The idea of lpLDA is to generate the stochastic information called limiter. When its speed is higher than the speed predicted by lpWDA, we know that lpWDA is being too aggressive in stealing slack from lower priority jobs and the limiting speed should be used. Therefore, limiting the slack utilized by higher priority tasks in lpWDA requires a careful trade-off between being aggressive and being conservative. Furthermore, to decrease the time complexity, lpLDA uses the deadline dicur of job J_i^{cur} rather than checking every scheduling point for the minimum constant speed.

The necessary modifications of lpLDA are marked by ※※※ in Algorithms 1 and 4. We refer to this addition to lpWDA as the Average Case Limiter (ACL) in lines 11 and 12. In Algorithm 1, we add line 3, which initializes the average number of cycles that must be completed before each job deadline. Lines 31, 33 and 38 in Algorithm 4 ensure that the current execution information is stored. Line 11 in Algorithm 1 computes the speed required by each job to meet its deadline on average. Finally, line 12 of Algorithm 1 selects the maximum of the speed requested by lpWDA and the limiter, especially restricting the amount of slack that lpWDA can use. Notably, the transition-aware version of lpLDA (i.e. the lpLDAT) contains an additional algorithm presented in [18]. In accordance with the Algorithms 1 and 4, the scheduling results produced by lpWDA and lpLDA are similar. The reason is that the speed selected by lpLDA is always greater than or equal to the speed selected by lpWDA [18]. In addition, the worst-case utilization of the task set mentioned in Table 1 is high (i.e., $\frac{11}{12}$), and the speed based on the limiter proposed by lpLDA does not higher than the speed predicted by lpWDA. Therefore, the limiting speed computed by lpLDA should not be used. In order to focus on how to collect more slack time in the schedule, the scheduling result produced by lpLDA is abridged.

2.4. Motivational Example

There is a bit of a different between lpLDA and lpFSA. lpLDA modifies lpWDA and becomes a new version of slack-time analysis method while lpFSA provides these methods (e.g. lpWDA, lpLDAT, lpWDA-AC, etc.) a subroutine to improve their ability of slack-time analysis. The main advantage is that lpFSA can be independent of each specific slack analysis method. For instance, the main purpose of lpWDA-AC and lpWDA-DP is to decrease the context-switch overhead while lpLDAT is to reduce the transition time and energy overhead. Other benefits of lpFSA are simple and good compatibility. In this paper, the methods compatible with lpFSA are called the *host* algorithms of lpFSA. Although lpWDA is a linear time slack analysis method, this heuristic estimates the available slack in the interval only up to the upcoming deadline of lower priority tasks.

TABLE I. AN EXAMPLE OF REAL-TIME TASK SET *T*

Task	Period(p_i)	WCET(wc_i)	ACET(ac_i)
τ_1	3	1.0	0.5
τ_2	4	1.0	0.5
τ_3	6	2.0	1.0

Example 1. Consider a periodic task set *T*, it presents the period length, WCET and ACET of each task in the Table 1. Figure 2(a) presents the execution schedule under the worst-case workload in the first hyperperiod. Figure 2(b) shows the speed schedule under the lpWDA algorithm for the task set *T* and assumes the actual *work* of each task is equal to its ACET. Before assigning $\tau_{1,1}$ at time t=0, lpWDA computes the available slack-time up to $d_{3,1}$=6 by calling Algorithm 3 recursively. However, interval [0, 6) has none of slack-time under the worst-case execution schedule. If we extend the length of analysis interval up to $2\times p_n$, one unit of slack-time is derived from $2 \times p_n - \sum_{i=1}^{n} \lfloor \frac{2p_n}{p_i} \rfloor \times wc_i$. One can imagine the slack in [11,12) as fluid, exchanging it with earlier work and moving it backward to the current scheduling point. For instance, in Figure 2(a), the slack in interval [11,12) can be exchanged with the work in interval [7,8), and then slack in interval [7,8) can be exchanged with the work in interval [4,5), and it can be exchanged once again with the work in interval [2,3). Finally, the slack in interval [2,3) can be exchanged with the work in interval [1,2). Therefore, in Figure 2(c), $\tau_{1,1}$ is scheduled with speed $S_{1,1} = \frac{wc_i}{wc_i+1}$. This example presents that an additional future slack can be utilized by current job and keeps the deadlines of the subsequent jobs. Actually, lpFSA does not move all of the jobs in the schedule to *readyQ* at once (e.g., t=0) or exchange the slack with work for using this slack. In the proposed method, most jobs are scheduled under RM priority policy and lpWDA.

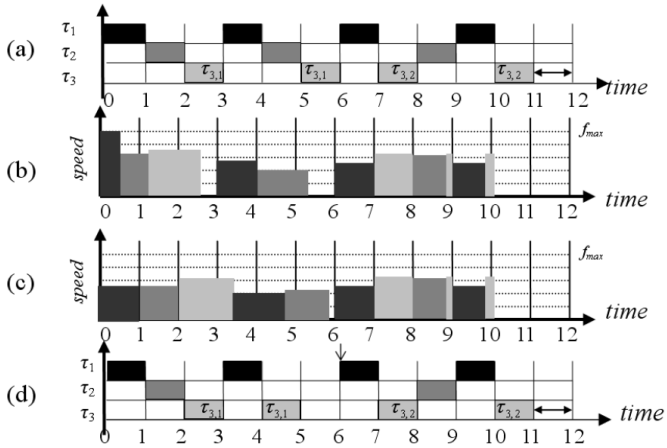

Figure 2. The intertask voltage scheduling examples of (a)worst-case scheduling, (b)lpWDA, (c)lpWDA+lpFSA and (d)a modified worst-case schedule.

Unfortunately, this straightforward idea cannot work in the actual situations. For example, in Figure 2(d), when p_2 is modified to 6, the slack in the interval [11, 12) cannot be transferred before t=6. In detail, jobs $\tau_{1,3}$, $\tau_{2,2}$ and $\tau_{3,2}$ release simultaneously at time 6. The slack in interval [11, 12) cannot be exchanged with the *work* of $\tau_{1,2}$, $\tau_{2,1}$ or $\tau_{3,1}$, because a deadline missing is likely to take place in one of those three jobs. Thus, this slack cannot shift to an earlier time in the schedule and improve power efficiency.

In this paper, our goal is to devise an efficient and more accurate slack analysis method for DVS RM scheduling. The idea in Example 1 lengthens the analysis interval for obtaining additional slack.

3. BASIC IDEA

Let $r_{n,k}$ denote the *border* of τ_i which is the first scheduled job at time t where $t \geqq r_{n,k-1}$, we computes the length of additional slack in the interval [*border*, $r_{n,k+1}$). For example, Figure 3(a) presents the scheduled task set mentioned in Table 1. When job $\tau_{1,1}$ is ready at time t=0, the current *border* is at $r_{3,2}$=6 and the target interval for extracting more slack time is [*border*, $r_{3,3}$). In this case, the period of $\tau_{2,2}$ *astrides* the *border* while the periods of $\tau_{1,2}$ and $\tau_{3,1}$ are exactly finished at the *border*. To precisely compute how much the additional slack can be transferred like liquid from interval [*border*, $r_{n,k+1}$) to [$r_{n,k-1}$, *border*), lpFSA has the following two phases.

Phase 1: In the interval [*border*, $r_{n,k+1}$), we compute the minimum available slack that can be shifted to approach the right-hand side of *border*.

Phase 2: Analyze the amount of slack that can be moved across the *border*.

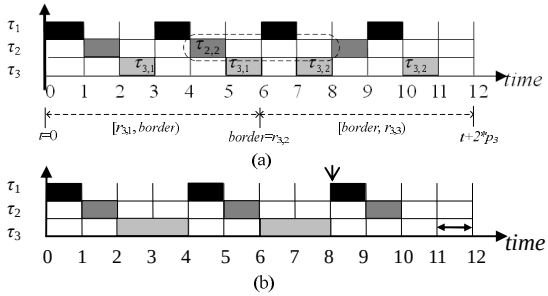

Figure 3. The DVS scheduling example.

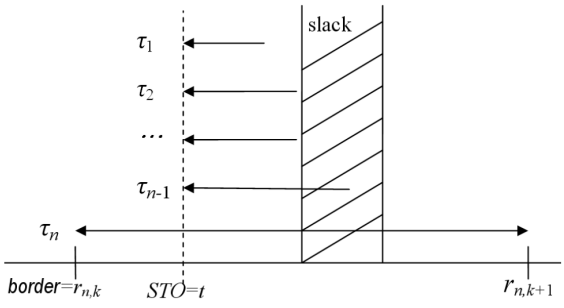

Figure 4. An example of *STO*.

As long as the slack shift to an earlier time than *border*, it can be utilized by an lpWDA-based method to improve the energy efficiency of the schedules. In the Phase 1, one may wonder why we only focus on the slack computation in the interval [*border*, $r_{n,k+1}$) but longer or shorter interval. Even if all the jobs except τ_n are within [*border*, $r_{n,k+1}$), they cannot make a target slack shift to the right-hand side of *border*. Job τ_n is still able to play the role of exchanging its work with the slack such that it can approach toward the *border*. For example, in Figure 3(b), when we change the period length of τ_1 from 3 to 4, the slack in interval [11, 12) cannot be exchanged with the work of $\tau_{1,3}$ or $\tau_{2,3}$ because it is hampered at time 8. So far, only $\tau_{3,2}$ can move the slack by exchanging with its *work* in [6, 7) to approach the right-hand side of *border*=6. On the contrary, if we extend the additional analysis interval longer than p_n, job τ_n cannot move such slack to approach the *border* and may be blocked in this interval. Therefore, to extend the analysis interval longer than $2\times p_n$ does not increase a substantial energy saving but increase the computing overheads during slack-time analysis. In the Phase 2, after deriving the amount of slack that has the potential to approach the *border*, the job periods astriding the *border* are applied to compute the available slack derived from phase1. The available slack is exchanged with the *work* of the jobs astriding the *border* and their work must be situated before *border*. After completing Phase 2, lpWDA-based methods can utilize the additional slack. In the next section, we present a linear-time heuristics algorithm for fluid slack analysis.

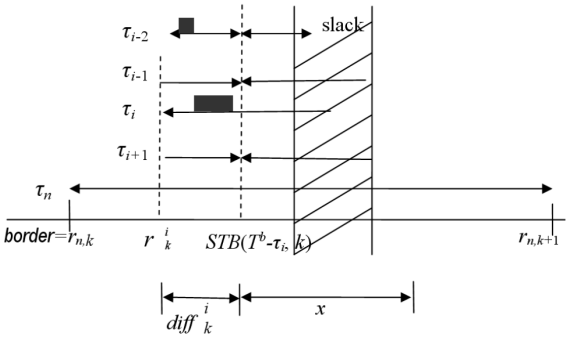

Figure 5. An example of *STB*.

4. LOW-POWER FLUID SLACK ANALYSIS(LPFSA)

Before presenting the slack computation method using the fluid slack analysis, we introduce the following notations.

$$T^b = T - \{\tau_n\}.$$

where b denotes the number of tasks in T^b and $b<n$ and τ_n denotes the task with the longest period in T. In an extended analysis interval [*border*, $r_{n,k+1}$), the number of synchronization points of the tasks in T^b is computed as follows:

$$Syn(T^b, k) = \left\lfloor \frac{r_{n,k+1}}{LCM\ (T^b)} \right\rfloor - \left\lfloor \frac{r_{n,k}}{LCM\ (T^b)} \right\rfloor . \quad (1)$$

where $LCM(T^b)$ denotes the least common multiple of the task periods in T^b. We define the tasks are *synchronous* at time *t*, their jobs release at time *t*. Therefore, the first *synchronous* point of T^b within the interval [*border*, $r_{n,k+1}$) is derived as

$$t(T^b,k)=\left\lceil \frac{r_{n,k+1}}{LCM(T^b)} \right\rceil \times LCM(T^b). \quad (2)$$

According to equation (1), two situations in interval [*border*, $r_{n,k+1}$) are defined as:

Slack Transmission Obstacle (STO):

$$Syn(T^b,k)>0, \quad b=n\text{-}1.$$

Slack Transmission Bottleneck (STB):

$$Syn(T^b,k)=0, \quad b=n\text{-}1.$$

When an *STO* appears in the additional analysis interval, the slack time is possibly likely to be blocked or diminished by the synchronous point produced by the tasks period in T^b. For example, in Figure 4, the tasks except τ_n are synchronized at time *t*. If a slack exists after time *t*, it cannot moves backward to the left-hand side of *t*. In this case, the slack can still be shifted by exchanging with the *work* of τ_n and be discussed later in phase 2. When $Syn(T^b,k)=0$ and *b*=*n*-1, the amount of fluid slack approaching the current *border* can be estimated by performing Phase 1. For example, in Figure 5, τ_i does not synchronize with other tasks in T^b. Therefore, we can compute the value of $Syn(T^x=T^b\text{-}\tau_i, k)$ for each τ_i where *x*=*n*-2 and *b*=*n*-1. Suppose $Syn(T^b-\tau_i,k)>0$, the earliest synchronization point of the tasks in $T^b\text{-}\tau_i$ is derived from equation (2). In interval [*border*, $r_{n,k+1}$), we define the earliest slack transmission bottleneck incurred by task set $T^b\text{-}\tau_i$ as follows:

$$STB(T^b-\tau_i,k)=t(T^b-\tau_i,k), \ i\neq n \text{ and } b=n\text{-}2. \quad (3)$$

The release time of τ_i astriding $Syn(T^b-\tau_i,k)$ is defined as

$$r_k^i=\left\lfloor \frac{STB(T^b-\tau_i,k)}{p_i} \right\rfloor \times p_i, \tau_i \notin T^b. \quad (4)$$

The difference between equations (3) and (4) is defined as follows:

$$diff_k^i=\max\{STB(T^b-\tau_i,k)-r_k^i, 0\}. \quad (5)$$

In Figure 5, suppose an initial slack is in the period of at least one task in T^b, the amount of slack that can be shifted across $STB(T^b-\tau_i,k)$ depends on the length of *work* of τ_i within interval [r_k^i, $r_k^i+diff_k^i$]. However, the higher priority tasks in $T^b\text{-}\tau_i$ may interfere with the length of work of τ_i in this interval. To precisely estimate the amount of *work* of τ_i in interval [*border*, $r_{n\ k+1}$), the higher priority *work* is classified into two parts. The first part of the *work* is provided by the tasks with period that shorter than or equal to $diff_k^i$. We define

$$H_i^{short}(\ell)=\sum_{\tau_k\in T^b-\tau_i}\left\lfloor \frac{\ell}{p_k} \right\rfloor \times wc_k \quad , \quad \ell>p_{i\text{-}1} \quad (6)$$

where ℓ denotes the length of $diff_k^i$. For example, in Figure 5, the value of $diff_k^i$ is ℓ and $p_{i\text{-}2}$ is shorter than ℓ. The worst-case execution cycles of job $\tau_{i\text{-}2}$ must be included in $H_i^{short}(\ell)$, because $\tau_{i\text{-}2}$ does not astride the border and has higher priority than that of τ_i. The second part is the additional work required by the task with periods not less than $diff_k^i$. That is

$$H_i^{long}(\ell)=\max\{(R_{i\text{-}1}+r_{i\text{-}1})-r_i, \ 0\}, \ell\leq p_{i\text{-}1} \quad (7)$$

where $R_{i\text{-}1}$ denotes the worst-case response time of $\tau_{i\text{-}1}$. Because RM scheduling, job τ_h with $\ell \leqq p_h<p_i$ has higher priority than that of τ_i. Moreover, since the period of τ_h does not astride the *border*, its *work* cannot exchange with the slack located in the right-hand side of *border*. By equations (6) and (7), the amount of higher priority work required in interval [r_k^i, $r_k^i+diff_k^i$) can be express as

$$H_i^{exec}(\ell)=\begin{cases} H_i^{short}(\ell)+H_i^{long}(\ell), & \ell\leq p_{i\text{-}1} \\ H_i^{short}(\ell) & \text{, otherwise.} \end{cases} \quad (8)$$

The value of $H_i^{exec}(\ell)$ denotes the length of work in $diff_k^i$ and cannot be exchanged with the slack at the right-hand side of *border*. In the worst case, τ_i is the only asynchronous job in T^b and $p_{i\text{-}1}\geqq diff_k^i$, we compute the work of the tasks τ_b ($b\leqq i$-1) of which their periods across r_i by computing the worst-case response time [11] of $\tau_{i\text{-}1}$. Therefore, the estimated *work* of τ_i in the interval [r_k^i, $r_k^i+diff_k^i$] is derived from

$$e_i^{Exchange}=\min\{\max\{diff_k^i-H_i^{exec}(diff_k^i),0\}, wc_i\}. \quad (9)$$

After completing Phase 1, Phase 2 computes the length of the slack that can be exchanged across the *border*.

Before transferring the slack to cross the *border*, we continue with the case of $Syn(T^b,k)$ =1 and *b*=*n*-1 mentioned in equation (1). When the worst-case response time of τ_n is not greater than $t(T^b, k)$, the slack situated after $t(T^b, k)$ can be shifted to the right-hand side of *border* by exchanging with a part of wc_n. That is,

$$e_n^{Exchange}=\begin{cases} wc_n & , \ R_n\leq t(T^b,k) \text{ and } b=n\text{-}1, \\ wc_n-(R_n-t(T^b,k)) & , \ R_n>t(T^b,k) \\ 0 & \text{, otherwise.} \end{cases} \quad (10)$$

In the equation (9), even if $e_i^{Exchange}$=0, we can utilize equation (10) to move the slack by using $e_n^{Exchange}$.

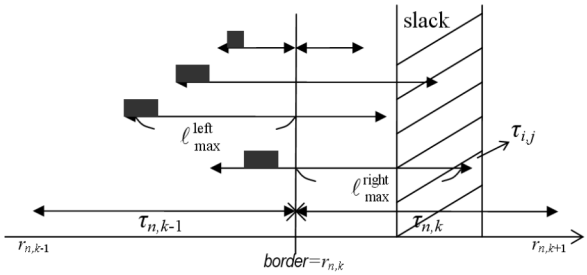

Figure 6. The task periods astride the *border*.

The remainder is to compute the amount of slack that can be transferred across the *border*. Assuming T^{border} denotes a task set in which the tasks *astride* the *border*. Let $\tau_i \in T^{border}$, the lengths of the *left* and the *right* portion of p_i

split by *border* is defined as ℓ_i^{left} and ℓ_i^{right}, respectively. The longest ℓ_i^{left} and ℓ_i^{right} is defined as $\ell_{\max}^{\text{left}}$ and $\ell_{\max}^{\text{right}}$, respectively. In addition, we define

$$accu^{\text{border}} = \sum_{\tau_i \in T^{\text{border}}} wc_i$$

as the total amount of work in the T^{border}. As shown in Figure 6, the lengths of $\ell_{\max}^{\text{left}}$, $\ell_{\max}^{\text{right}}$ and $accu^{\text{border}}$ limit the maximum amount of slack that can be transferred across *border*. Consequently, the restriction on the amount of slack in Phase2 can be described as

$$e^{\text{border}} = \min\left\{\ell_{\max}^{\text{left}}, \ell_{\max}^{\text{right}}, accu^{\text{border}}\right\}. \quad (11)$$

After completing Phase 1 and Phase 2, we derive the amount of slack which can approach and cross the *border*. In the interval $[r_{n,k-1}, r_{n,k+1})$, the available slack can be estimated as

$$e^{\text{slack}} = 2 \times p_n - \sum_{\tau_i \in T} \left\lceil \frac{2 \times p_n}{p_i} \right\rceil wc_i. \quad (12)$$

Based on the equations mentioned above, the algorithm of the fluid slack analysis is presented in Figure 7.

Procedure:lpFSA(time t, task set T)
Input t :present time, $T^b = T - \tau_n$

01. **set** $b = n-1$, $e_{\min}^{\text{Exchange}} \leftarrow \infty$, $\varepsilon \leftarrow 0$, $k = \lceil \frac{t+\varepsilon}{p_n} \rceil$, $\ell \leftarrow 0$.
(Phase1)
02. **if** $Syn(T^b, k) := 0$
03. **for** $i := 1$ **to** $n-1$
04. **if** $Syn(T^b - \tau_i) := 0$ **then** continue
05. **if** $\ell < diff_k^i$ **then**
06. $\ell := diff_k^i$ and $\tau_{\text{asyn}} := \tau_i$
07. **if** $\ell \leq$ **then** $\tau_{asyn} := \emptyset$
08. Compute the value of $e_{\text{asyn}}^{\text{Exchange}}$.
(Phase 2)
09. **else if** $Syn(T^b, k) > 0$ **or** $e_{\text{asyn}}^{\text{Exchange}} \leqq 0$
10. **then** Compute the values of e_n^{Exchange}, e^{border} and e^{slack}
11. **if** $e_{asyn}^{\text{Exchange}} \leqq 0$ **then** $e_{\min}^{\text{Exchange}} := \min\{e_n^{\text{Exchange}}, e^{\text{border}}, e^{\text{slack}}\}$
12. **else** $e_{\min}^{\text{Exchange}} := \min\{e_n^{\text{Exchange}} + e_{asyn}^{\text{Exchange}}, e^{\text{border}}, e^{\text{slack}}\}$
13. Set τ_{asyn} as the lowest priority job τ_δ in T^{borter} with $\ell_\delta^{\text{right}}$ not shorter than $e_{\min}^{\text{Exchange}}$;
Return $e_{\min}^{\text{Exchange}}$

Figure 7. The algorithm lpFSA.

Example 2. Consider the example of WCET schedule shown in Figure 2(a). Before assigning $\tau_{1,1}$ at time t=0, we can derive *border*=6 and $T^{\text{border}} = \{\tau_{2,2}\}$ according the task periods in T. Procedure lpFSA can estimate the length of fluid slack from interval [6, 12) as follows. When task set $T^2 = \{\tau_1, \tau_2\}$, Procedure lpFSA computes Syn(T^2,1)=0. Therefore, in Phase1, the bottleneck caused by τ_1 and τ_2 is $STB(T^2 - \tau_2, 1) = 9$ and $STB(T^2 - \tau_1, 1) = 8$,respectively.

In line 6, we can derive ℓ=2 and $\tau_{asyn} = \tau_1$. According to equation (7), (8) and (9), we derive $e_{asyn}^{\text{Exchange}}$=1. In line 10, the value of e_n^{Exchange}, e^{border}, e^{slack} and $accu^{\text{border}}$ is 1, 1, 1 and 2, respectively. The value of $e_{min}^{\text{Exchange}}$ is one because of line 12. Finally, τ_{asyn} is $\tau_{2,2}$ and becomes the lowest priority job among $T^{\text{border}} \cup \tau_{n,k}$. Therefore, algorithm lpFSA in Figure 7 returns $e_{min}^{\text{Exchange}}$=1 to the Algorithm lpWDA and passes additional slack e^{Exchange} to Algorithm CalcLowerPriorityWork(). Notably, the tasks using lpFSA still execute under RM priority policy except one of the jobs whose periods span astride the border. At time t=0, when jobs $\tau_{1,1}$, $\tau_{2,1}$ and $\tau_{3,1}$ enter *readyQ* at time t=0, $\tau_{1,1}$ has the highest priority and utilizes additional slack $e_{\min}^{\text{Exchange}}$ estimated by lpFSA. Therefore, job $\tau_{1,1}$ obtains one unit of time of slack and changes its voltage level from 1 to 0.5. On the contrary, if primitive lpWDA performs $\tau_{1,1}$ at time t=0, $\tau_{1,1}$ cannot obtain any slack. When lpWDA executes iteratively, the value of e^{Exchange} does not change until $\tau_{1,1}$ is completed. Figure 1(c) presents the scheduling result obtained using Procedure lpFSA. After completing $\tau_{1,1}$, $e_{\min}^{\text{Exchange}}$ unit of slack has been run out, primitive lpWDA continuously performs voltage scaling on the subsequent jobs of $\tau_{1,1}$. In the case of $\tau_{2,1}$, it begins after $\tau_{1,1}$ (t=1) and obtains one unit of slack time from primitive lpWDA. Therefore, its WCET under voltage v=0.5 is changed to $wc_{2,1}$= 2 and actual execution time is $ac_{2,1}$ = 1. At time t=4, job $\tau_{2,2}$ is released and moved to *readyQ*. Its priority is changed to and lower than the remaining execution time of $\tau_{3,1}$ by executing line 14 in Procedure lpFSA. Therefore, job $\tau_{2,2}$ begins its work after completing the remaining work of $\tau_{3,1}$. Notably, lpFSA only changes job's priority in T^{border} and does not affect the feasibility of lpWDA schedule.

The scheduling result using lpFSA is presented in Figure 2(c), and the values of scheduling parameters are shown in Table 2. Job τ_{asyn} is a global variable mentioned in Algorithm 1. Whenever job τ_{asyn} executes and $e^{\text{Exchange}} > 0$, it lowers its priority to guarantee its timing constraint of job τ_n.

TABLE II. SCHEDULING PARAMETERS IN EXAMPLE 2

time	*readyQ*(t)	ud_γ	r	e^{Exchange}	slack	voltage	wc	ac
0	$\rightarrow \tau_{3,1}, \tau_{2,1}, \tau_{1,1} \rightarrow$	6	3	1	1	$\frac{1}{1+1} = 0.5$	2	1
1	$\rightarrow \tau_{3,1}, \tau_{2,1} \rightarrow$	6	3	1	1	$\frac{1}{1+1} = 0.5$	2	1
2	$\rightarrow \tau_{3,1} \rightarrow$	6	3	1	1	$\frac{2}{2+1} = 0.67$	3	1.5
3	$\rightarrow \tau_{3,1}, \tau_{1,2} \rightarrow$							
3.5	$\rightarrow \tau_{1,2} \rightarrow$	6	3	1	1.5	$\frac{1}{1+1.5} = 0.4$	2.5	1.25
4.75	$\rightarrow \tau_{2,2} \rightarrow$	12	3	0	1.25	$\frac{1}{1+1.25} = 0.44$	2.25	1.125
5.875	$\rightarrow \rightarrow$							
6	$\rightarrow \tau_{3,2}, \tau_{1,3} \rightarrow$							

Theorem 3. *Algorithm* lpFSA *has a computational complexity of O*(*n*) *per scheduling point, where n denotes the number of tasks in the systems.*

Proof. In the phase 1, lines 4 and 5 are completed in constant time for each iterative step according to equations (2), (3) and (4). In the line 8, the value of $e_{asyn}^{Exchange}$ is derived from equations (6), (7),(8) and (9), where the value of $H_i^{short}(\ell)$ in equation (5) needs O(*n*) time to compute the

amount of work. The worst-case response time R_i of task τ_i in equation (7) is computed by accumulating those of previous tasks. In the line 10 of phase 2, the values of $e_n^{Exchange}$, e^{border} and e^{slack} are derived in O(n) time. Therefore, the overall time complexity is O(n).

5. PERFORMANCE EVALUATION

In this section, we evaluate the effectiveness of lpFSA on randomly generated task sets and compare its energy consumption with ccRM, lpWDA and lpLDAT. Both ccRM and lpWDA are modified to account for transition time overhead. In the simulations, lpWDA and lpLDAT are called the *host* algorithms of lpFSA to compare with initial algorithms ccRM, lpWDA and lpLDAT.

In the simulations, each task is characterized by its worst-case execution wc_i, its period p_i and its deadline d_i, where d_i=p_i. We vary three parameters in our simulations: (1) number of tasks *totaltasks* in T from 2 to 20 in two task increments, (2) utilization U for task set, from 0.1 to 0.9 and (3) the ratio ac/wc of ACET to WCET, from 0.1 to 0.9. For any given pair of *totaltasks*, U and ac/wc in T, we randomly generate 1000 task sets, and the experiment result is the average value over these 1000 task sets. In a task set, every task period p_i (and deadline d_i) is uniformly distributed in the range [1, 100]ms, each schedule is less than or equal to five hyperperiods in length. The execution time wc_i of each task is assigned in the real number range [1, min{p_i-1, 90}]ms. After giving the values of tasks' periods and executions, we assign the utilization U of a task set and rescale the p_i of each task such that the summation of the task weights (i.e. wc_i/p_i) is equal to a given U. The early completion time of each job in simulation (1) and (2) was randomly drawn from a Gaussian distribution in the range of [BCET, WCET], where BCET/WCET=0.1. In the simulation (3), each experiment was performed by varying BCET from 10% to 90% of WCET.

The processor model we assumed is based on the ARM8 microprocessor core. For all experiments, we assume there are 10 frequency levels available in the range of 10 to 100MHz, with corresponding voltage levels of 1 to 3.3 Volts. The energy consumptions of all the experiment results are normalized against the same as the processor running at maximum speed without DVS technique.

In the Figure 8, we examine the execution time that is required of each online algorithm, including:

a) ccRM: The algorithm ccRM from [19] is modified to account for transition overhead.

b) lpWDA: The algorithm lpWDA from [11] is modified to account for transition overhead.

c) lpLDA: The algorithm lpLDAT from [18].

d) lpWDA-lpFSA: The proposed algorithm is performed in conjunction with host algorithm lpWDA.

e) lpLDAT-lpFSA: The proposed algorithm is performed in conjunction with host algorithm lpLDAT.

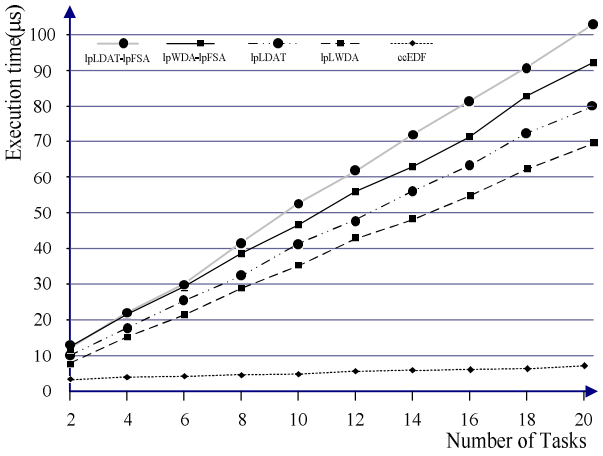

Figure 8. Maximum execution time for the scheduling algorithms versus the number of tasks on a 100MHz processor.

Figure 8 presents the maximum execution time of each algorithm on the target processor versus the number of tasks in the system. The result was generated by inserting a system timer function and executing each algorithm separately. Obviously, ccRM has a large advantage with respective to time complexity when compared to other online algorithms. Because the algorithms are invoked upon each release and completion, it is necessary to increase the execution time of each task by two times the maximum execution time of the algorithm to account for the scheduling overhead. To measure the execution time of the scheduling algorithms, we introduce additional assumptions as follows. First, the set of experiments present the execution time of lpFSA and its host algorithms(lpWDA and lpLDAT). We use the functions of system timer to record the duration of each algorithm, choose their longest execution times in each schedule, and accumulate the execution times separately. At the end of the experiment, the number of generated schedules divides the accumulated execution times. Figure 7 illustrates the maximum execution times of each algorithm on the CPU with highest speed(100MHz) versus the number of tasks in the system. lpFSA is an efficient on-line algorithm, which increases additional execution time less than 45% of those of their host algorithms.

The overheads considered in the simulations are as follows.

a) Algorithm execution time and energy The execution time of each algorithm refers to the simulation results in Figure 8. Its energy overhead is obtained under the assumption of the maximum speed S_{max}.

b) Voltage transition time and energy

The assumption of voltage scaling overhead is the same as that in [3], For the voltage scaling from V_{dd1} to V_{dd2}, the transition time is:

$$\Delta t \approx \frac{2 \cdot C}{I_{\max}} \cdot |V_{dd\,2} - V_{dd\,1}|$$

where C and $I_{\max}$ denote the charge to the capacitor and the maximum output current of the converter. The transition time is at most 70us between maximum transition [4]. The energy consumed during each transition is :

$$\Delta E = n(1-\lambda) \cdot C \cdot |V_{dd\,2} - V_{dd\,1}|$$

where λ denotes the efficiency of DC-DC converter.

c) *Context-switch time and energy*
The context-switch is assumed to be 50μs at the highest speed $S_{\max}$ as presented by David in [7].

The average energy consumptions of lpFSA are in the Figures 9, 10 and 11. Notably, counting these energy consumptions include not only execution duration of lpFSA and its *host* (i.e., lpWDA and lpLDA) algorithms but also the context-switching time to and from other real-time tasks. Since the range of task periods has been shortened between [1, 100]ms, the difference between task periods and context-switch times or transition times are smaller than those assumed in [11, 18]. In addition, these energy overheads arose from lpFSA and its host algorithm are also being taken into account, the experimental results can be closer to the actual situations. In these simulations, the *host* algorithms with lpFSA are still better than their initial algorithms respectively.

Figures 9, 10 and 11 also include the results for a clairvoyant algorithm, named bound, which knows the exact actual execution cycle of each task beforehand and adopts an optimal speed accordingly. Every scheduling point in the whole schedule is checked when looking for the best start and finish times, and the transition time is always considered zero. In fact, bound is not a practical algorithm because it is extremely time-consuming for finding the suitable start, preemption and completion times, and no algorithm predict the exact execution cycles beforehand. It plays a yardstick in the simulations because no real DVS algorithm can provide better performance than that of bound.

As shown in Figure 9, lpWDA-lpFSA and lpLDAT-lpFSA reduces the energy consumption up to 18% and 15% over lpWDA and lpLDAT, respectively. The utilization of a given task set is assigned randomly from 10% to 90% by a uniform probability distribution function. As the number of tasks from 4 to 12, the energy consumption of lpWDA-lpFSA and lpLDAT-lpFSA are increased steadily. The reason for this fact is that lpFSA focuses on exploiting the slack in the extended analysis scope. When the number of task increases, the number of *STO* and *STB* in the analysis interval is likely to decrease and benefits the computation of slack time.

The experimental results in Figure 10 present that lpWDA-lpFSA and lpLDAT-lpFSA perform 10% and 5% more energy saving compared to lpWDA and lpLDAT. In the experiment, the *totaltasks* of each task set was randomly determined from 2 to 20. As the utilization of the task set decreases, the energy saving of lpWDA-lpFSA and lpLDAT-lpFSA increase substantially while those of ccRM, lpWDA and lpLDAT are not. The explanation of the results is that lpFSA exploits the slack in an additional interval.

In Figure 11, clearly lpWDA-lpFSA and lpLDAT-lpFSA also outperform the other three algorithms in all cases, and the improvement increases steadily as BCET/WCET decreases.

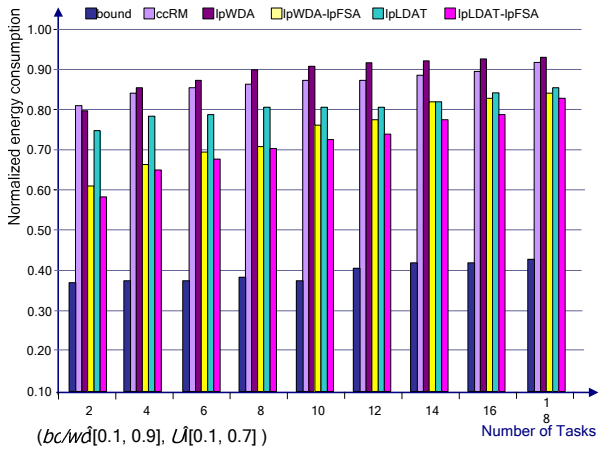

Figure 9. Energy consumption under different size of task sets.

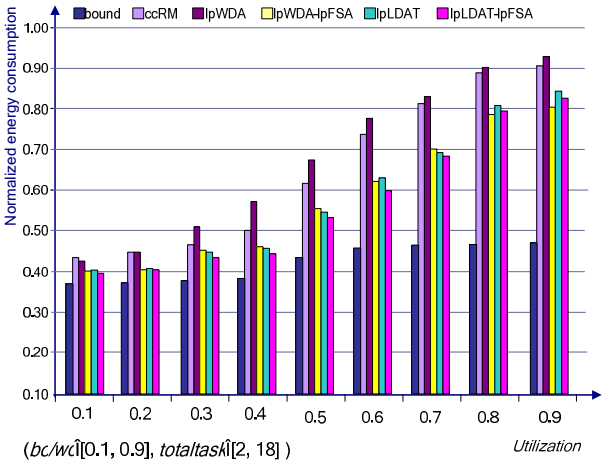

Figure 10. Energy consumption under different utilization.

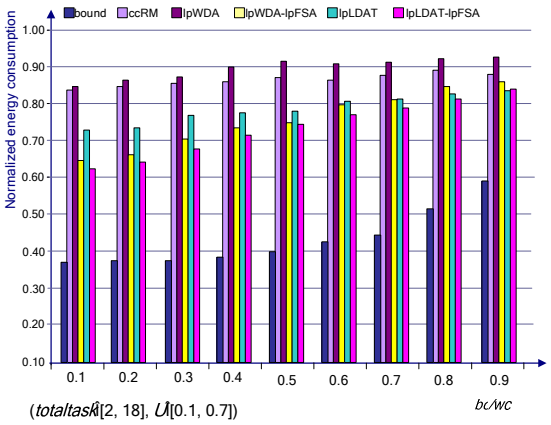

Figure 11. Energy consumption under different ratio of BCET to WCET sets.

6. CONCLUSIONS

In this paper, we proposed an on-line DVS algorithm based on the concept of fluid slack analysis called lpFSA. This algorithm is the first in its class that can be built in the bottom of the existing RM DVS methods for decreasing the power consumptions without increasing their time complexities. Our experimental results show that lpFSA can reduce overall energy consumption up to 45% when compared to initial methods.

Several directions will prove worthy for future work. Although this work focused on RM scheduling, the proposed fluid slack analysis can be applied to other scheduling policies such as earliest-deadline first (EDF) and EDF* [2]. Additionally, the existence of *STO* in the lpFSA hampers the transmission of slack in an analysis interval. Future work can prevent the *STO*s by relaxing job release times, thereby increasing available slack.

ACKNOWLEDGMENT

The authors acknowledge support from research grants from R.O.C. National Science Council NSC-99-2221-E-146-011.

REFERENCES

[1] AMD, "Mobile amd athlon 4 processor model 6 cpga data sheet rev:g," Advanced Micro Devices, *Technique Report 24332*, October 2003. [1]

[2] H. Aydin, R. Melhem, D. Mosse and P. Mejia-Alvarez. Power-Aware Scheduling for Periodic Real-Time Tasks. *IEEE Trans. Comput.*, 53(5): 584-600, May 2004.

[3] T. D. Burd and R. W. Bordersen, "Design issue for dynamic voltage scaling," *International Symposium on Low-power Electronics and Design*, 2000, pp. 9-14.

[4] T. D. Burd, T. Pering, A. Stratakos, R. Brodersen, "A dynamic voltage scaled microprocessor system," *IEEE journal of Solid-State Circuits*, vol, 35, No. 11, November 2000, pp. 1571-1580.

[5] Da-Ren Chen and Chiun-Chieh Hsu, "Transition-Aware Dynamic Voltage Scaling for Jitter-Controlled Real-Time Scheduling: A Tree-Structured Approach," *in the 38th international conference on Parallel Processing* (*ICPP'09*), 2009.

[6] Jian-Jia Chen and Chin-Fu Kuo, "Energy-Efficient Scheduling for Real-Time Systems on Dynamic Voltage Scaling (DVS) Platforms," *in the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*(*RTCSA 2007*), Aug. 2007, pp.28-38.

[7] F. David, Jeffrey Carlyle and Roy Campbell, "Context-switch overheads for Linux on ARM platforms," *Experimental Computer Science* 2007:3.

[8] F. Gruian, "Hard real-time scheduling for low-energy using stochastic data and dvs processors," in Proceedings of the 2001 International Symposium on Low Power Electronics and Design (ISPLED'01). Huntington Beach, CA: ACM Press, Aug. 2001, pp.46-51.

[9] Xiao Chuan He and Yan Jia, "Energy-Efficient Scheduling Fixed-Priority Tasks with Preemption Thresholds on Variable Voltage Processors," *Lecture Notes in Computer Science*, *Springer Berlin/Heidelberg*, vol.4672, pp.133-142, 2008.

[10] Intel, "The intel xscale microarchitecture," *Intel Corporation, Technique Report*, 2000.

[11] W. Kim, J. Kim, and S. L. Min, "Dynamic Voltage Scaling Algorithm for Fixed-Priority Real-Time Systems Using Work-Demand Analysis," in *Proceedings of the 2003 International Symposium on Low Power Electronics and Design (ISPLED)*. New York, NY: ACM Press, Aug. 2003, pp. 396-401.

[12] Kim W, Shin D, Yun H, Kim J, Min S, "Preemption-Aware Dynamic Voltage Scaling in Hard Real-Time Systems," *in the Proceedings of the 2004 International Symposium on Low Power Electronics and Design (ISPLED)*. New York, NY: ACM Press, 2004 pp.393 – 398.

[13] John P. Lehoczky and Sandra Ramos-Thuel, "An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems," *in the Real-Time Systems Symposium(RTSS'92)*, 1992, pp.110-123.

[14] John P. Lehoczky, L. Sha and Y. Ding, "The Rate-Monotonic Scheduling Algorithm" *in the Real-Time Systems Symposium (RTSS'89)*, 1989, pp.166-171.

[15] John P. Lehoczky, "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines *in the Real-Time Systems Symposium (RTSS'90)*, 1990, pp.201-209.

[16] J. W. S. Liu, *Real-Time Systems*. Upper Saddle River, NJ: Prentice Hall, 2000.

[17] Jacob R. Lorch and Alan Jay Smith, "PACE: a new approach to dynamic voltage scaling" *IEEE Trans. Computers*, Vol. 53, No. 7, pp.856-869, July 2004.

[18] Bren Mochocki, Xiaobo Sharon Hu, Gang Quan, "Practical On-line DVS Scheduling for Fixed-Priority Real-Time Systems, " *in Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium* (*RTAS'05*), 2005, pp.224-233.

[19] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low power embedded operating systems," in *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP)*. New York, NY: ACM Press, 2001, pp. 89-102.

[20] G. Quan and X. S. Hu, "Energy efficient Fixed-priority scheduling for real-time systems on variable voltage processors," *in the Proceedings of the 2001 Design Automation Conference (DAC)*. New York, NY: IEEE, June 2001, pp. 828.833.

[21] Samsung, "Samsung and Intrinsity Jointly Develop the World's Fatest ARM® CORTEX™-A8 Processor Based Mobile Core In 45 Nanometer Low Power Processor," **http://www.samsung.com/**.

[22] D. Shin, S. Lee, and J. Kim, "Intra-task voltage scheduling for low-energy hard real-time applications," *Design & Test of Computers*, vol. 18,no. 2, pp. 20.30, March . April 2001.

[23] M.Weiser, B.Welch, A. J. Demers, and S. Shenker, "Scheduling for reduced CPU energy," *in Operating Systems Design and Implementation*, pages 13–23, 1994.