# Performance Comparison of Dynamic Voltage Scaling Algorithms for Hard Real-Time Systems

Woonseok Kim* Dongkun Shin† Han-Saem Yun† Jihong Kim† Sang Lyul Min*

School of Computer Science and Engineering
Seoul National University ENG4190, Seoul, Korea, 151-742
wskim@archi.snu.ac.kr, {sdk, hsyun, jihong}@davinci.snu.ac.kr, symin@dandelion.snu.ac.kr

## Abstract

*Dynamic voltage scaling (DVS) is an effective low-power design technique for embedded real-time systems. In recent years, many DVS algorithms have been proposed for reducing the energy consumption of embedded hard real-time systems. However, the proposed DVS algorithms were not quantitatively evaluated under a unified framework, making it a difficult task to select an appropriate DVS algorithm for a given application/system. In this paper, we compare several key DVS algorithms recently proposed for hard real-time periodic task sets, analyze their energy efficiency, and discuss the performance differences quantitatively. Our evaluation results give quantitative answers to several important DVS questions.*

## 1 Introduction

Dynamic voltage scaling (DVS), which adjusts the supply voltage and correspondingly the clock frequency dynamically, is an effective low-power design technique for embedded real-time systems. Since the energy consumption E of CMOS circuits has a quadratic dependency on the supply voltage $V_{dd}$, lowering the supply voltage $V_{dd}$ is one of the most effective ways of reducing the energy consumption.

With a recent explosive growth in the portable and mobile embedded device market, where a low-power consumption is an important design requirement, several commercial variable-voltage microprocessors [19, 1, 8] were developed. Targeting these microprocessors, many DVS algorithms have been proposed or developed, especially for hard real-time systems [7, 9, 18, 2, 14, 16, 5, 10]. Since lowering the supply voltage also decreases the maximum achievable clock speed [15], various DVS algorithms for hard real-time systems have the goal of reducing supply voltage dynamically to the lowest possible level while satisfying the tasks' timing constraints.

Although each DVS algorithm is shown to be quite effective in reducing the energy/power consumption of a target system under its own experimental scenarios, these recent DVS algorithms have not been quantitatively evaluated under a unified framework, making it a difficult task for low-power embedded system developers to select an appropriate DVS algorithm for a given application/system. A quantitative analysis of the energy-efficiency is particularly important because most of these DVS algorithms are based on both static and dynamic slack analysis techniques whose performance is difficult to predict analytically. In addition, their energy efficiency fluctuate significantly depending on the workload variations, task set characterizations, and execution paths taken, further requiring a quantitative comparison study.

In this paper, we quantitatively evaluate the energy efficiency of several recent DVS algorithms proposed for hard real-time systems using a unified DVS simulation environment called *SimDVS* [17]. We focus on a preemptive hard real-time systems in which periodic real-time tasks are scheduled with the Earliest-Deadline-First (EDF) algorithm or the Rate-Monotonic (RM) algorithm, the two most widely used real-time system models [12]. Our study is different from the previous performance comparisons such as [13, 6]. [13] and [6] focus on *aperiodic* tasks in hard real-time systems and non real-time systems, respectively, while our study focuses on periodic tasks in hard real-time systems.

For the target hard real-time systems, two categories of algorithms are used: inter-task DVS (InterDVS) and intra-task DVS (IntraDVS). InterDVS algorithms determine the supply voltage on task-by-task basis, while IntraDVS algorithms

*This work was supported in part by the Ministry of Education under the BK21 program, and by the Ministry of Science and Technology under the National Research Laboratory program.

†This work was supported by grant No. R01-2001-00360 from the Korea Science & Engineering Foundation.

adjust the supply voltage within an individual task boundary. For a comparative study, we use eight InterDVS algorithms [18, 2, 14, 10] and two IntraDVS algorithms [16, 5] that were recently proposed.

We also evaluate the energy efficiency of HybridDVS algorithms. (If a DVS algorithm uses both the IntraDVS and InterDVS approaches, we call the algorithm a hybrid DVS algorithm (HybridDVS).)

Since many factors affect the energy efficiency of DVS algorithms, our comparative study cannot answer all the DVS performance questions. In this paper, we limit our evaluation goals to the following questions which represent some of the most important unanswered questions:

- **InterDVS**: What is the best InterDVS algorithm under given conditions? How close is the algorithm's energy efficiency to the theoretical lower bound? What restrictions of variable-voltage processors, if any, limit the achievable energy efficiency of InterDVS algorithms?
- **IntraDVS**: Which IntraDVS algorithm performs better under what condition?
- **HybridDVS**: Can we achieve better energy efficiency if we combine an InterDVS algorithm and an IntraDVS algorithm?

Our comparative study shows that the existing EDF InterDVS algorithms such as [2, 14, 10], are very effective; their energy consumption is only 9$\sim$12% worse than the theoretical lower bound. Moreover, this gap can be further reduced by using a more intelligent slack distribution method. With a better slack distribution heuristic, we strongly believe that the energy efficiency of the current state-of-art EDF InterDVS algorithms is very close to that of the theoretical optimal algorithm. However, in the RM InterDVS algorithms, there still remains a room for improvement. Also, the energy efficiency of each algorithm can vary from 10% to 32% according to the number of voltage levels supported by the target variable-voltage processor.

For the IntraDVS algorithms, our results indicate that the path-based IntraDVS [16] achieves better performance than the stochastic IntraDVS [5] when the slack time is limited. On the other hand, when there is a large amount of slack time, the stochastic IntraDVS algorithm works better.

For the HybridDVS algorithms, our experiments show that the energy efficiency of a HybridDVS is better than the one that can be achieved by using an IntraDVS algorithm or an InterDVS algorithm alone.

The rest of the paper is organized as follows; before the selected DVS algorithms are evaluated, we first classify existing DVS techniques in Section 2. In Section 3, we summarize the selected DVS algorithms using the classification framework of Section 2. Simulation environments are described in Section 4. We present the performance evaluation results in Section 5, and Section 6 concludes with a summary.

**Table 1. Classification of DVS techniques.**

| | Voltage Scaling Methods | Scaling Decision |
|---|---|---|
| IntraDVS | (1) Path-based method | Off-Line |
| | (2) Stochastic method | |
| InterDVS | (3) Maximum constant speed | |
| | (4) Stretching to NTA | On-Line |
| | (5) Priority-based slack-stealing | |
| | (6) Utilization updating | |

## 2 Classification of DVS algorithms

In this section, we classify the existing DVS techniques and briefly describe the key characteristics of each technique. (See Table 1 for summary.)

For hard real-time systems, there are two kinds of voltage scheduling approaches depending on the voltage scaling granularity: intra-task DVS (IntraDVS) and inter-task DVS (InterDVS). The intra-task DVS algorithms [16, 5] adjust the voltage within an individual task boundary, while the inter-task DVS algorithms determine the voltage on a task-by-task basis at each scheduling point. The main difference between them is whether the slack times are used for the current task or for the tasks that follow. InterDVS algorithms distribute the slack times from the current task for the following tasks, while IntraDVS algorithms use the slack times from the current task for the current task itself.

### 2.1 Intra-task DVS algorithm design factors

In scheduling hard real-time tasks, in order to guarantee the timing constraint of each task, the execution times of tasks are usually assumed to be the worst case execution times (WCETs). However, since a task has many possible execution paths, there are large execution time variations among them. So, when the execution path taken at run time is not the worst case execution path (WCEP), the task may complete its execution before its WCET, resulting in a slack time. In that case, IntraDVS exploits such slack times and adjusts the processor speed. IntraDVS algorithms can be classified into two types depending on how to estimate slack times and how to adjust speeds.

#### 2.1.1 Path-based method

In the path-based IntraDVS, the voltage and clock speed are determined based on a predicted reference execution path, such as WCEP. For example, when the actual execution deviates from the predicted reference execution path (say, by a branch instruction), the clock speed is adjusted. If the new path takes significantly longer to complete its execution than

the reference path, the clock speed is *raised* to meet the deadline constraint. On the other hand, if the new path can finish its execution earlier than the reference path, the clock speed is *lowered* to reduce the energy consumption.

In the path-based IntraDVS, program locations for possible speed scaling are identified using static program analysis [16] or execution time profiling [11].

#### 2.1.2 Stochastic method

The stochastic method is based on the idea that it is better to start the execution at a low speed and accelerate the execution later when needed than to start with a high speed and reduce the speed later when slack time is found. By starting at a low speed, if the task finishes earlier than its WCET, it does not need to execute at a high speed. Theoretically, if the probability density function of execution times of a task is known *a priori*, the optimal speed schedule can be computed [5]. Under the stochastic method, the clock speed is *raised* at specific time instances, regardless of the execution paths taken. Unlike the path-based IntraDVS that can utilize all the slack times of the task in scaling speed, the stochastic IntraDVS may not utilize all the potential slack times.

### 2.2 Inter-task DVS algorithm design factors

InterDVS algorithms exploit the "run-calculate-assign-run" strategy to determine the supply voltage, which can be summarized as follows: (1) run a current task, (2) when the task is completed, calculate the maximum allowable execution time for the next task, (3) assign the supply voltage for the next task, and (4) run the next task. Most InterDVS algorithms differ during step (2) in computing the maximum allowed time for the next task $\tau$ which is the sum of WCET of $\tau$ and the slack time available for $\tau$.

A generic InterDVS algorithm consists of two parts: slack estimation and slack distribution. The goal of the slack estimation part is to identify as much slack times as possible while the goal of the slack distribution part is to distribute the resulting slack times so that the resulting speed schedule is as uniform as possible. Slack times generally come from two sources; *static slack times* are the extra times available for the next task that can be identified statically, while *dynamic slack times* are caused from run-time variations of the task executions.

#### 2.2.1 Slack estimation methods

**(1) Static slack estimation**

**Maximum constant speed** One of the most commonly used static slack estimation methods is to compute the *maximum constant speed*, which is defined as the lowest possible clock speed that guarantees the feasible schedule of a task set [18]. For example, in EDF scheduling, if the worst case processor utilization (WCPU) $U$ of a given task set is lower than 1.0 under the maximum speed $f_{max}$, the task set can be scheduled with a new maximum speed $f'_{max} = U \cdot f_{max}$. Although more complicated, the maximum constant speed can be statically calculated as well for RM scheduling [18, 5].

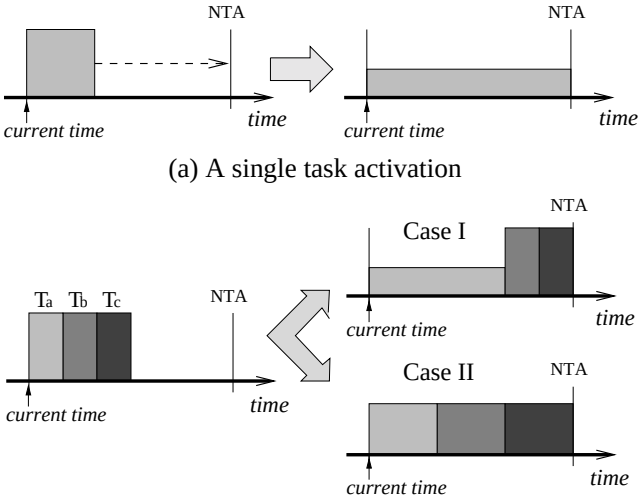



**Figure 1. Examples of Stretching-to-NTA.**

**(2) Dynamic slack estimation**

Three widely-used techniques of estimating dynamic slack times are briefly described below.

**Stretching to NTA** Even though a given task set is scheduled with the maximum constant speed, since the actual execution times of tasks are usually much less than their WCETs, the tasks usually have dynamic slack times. One simple method to estimate the *dynamic slack time* is to use the arrival time of the next task [18]. (The arrival time of the next task is denoted by NTA.) Assume that the current task $\tau$ is scheduled at time $t$. If NTA of $\tau$ is later than ($t$+WCET($\tau$)), task $\tau$ can be executed at a lower speed so that its execution completes exactly at the NTA.

Figure 1 shows examples of the *Stretching-to-NTA* method. When a single task $\tau$ is activated as shown in Figure 1(a), the execution of $\tau$ can be stretched to NTA. When multiple tasks are activated, there can be several alternatives in stretching options. For example, the dynamic slack time may be given to a single task or distributed equally to all activated tasks. Cases I and II of Figure 1(b) illustrate these two options, respectively.

**Priority-based slack stealing** This method exploits the basic properties of priority-driven scheduling such as RM and EDF. The basic idea is that when a higher-priority task completes its execution earlier than its WCET, the following

**Table 2. Target DVS algorithms.**

| Category | Scheduling Policy | DVS Policy | Used Methods† |
|---|---|---|---|
| InterDVS | EDF | lppsEDF [18] | (3)+(4) |
| | | ccEDF [14] | (6) |
| | | laEDF [14] | (6)* |
| | | DRA [2] | (3)+(4)+(5) |
| | | AGR [2] | (4)*+(5) |
| | | lpSHE [10] | (3)+(4)+(5)* |
| | RM | lppsRM [18] | (3)+(4) |
| | | ccRM [14] | (3)+(4)* |
| IntraDVS | Path-based Method | intraShin [16] | (1) |
| | Stochastic Method | intraGruian [5] | (2) |

† Numbers indicate corresponding techniques in Table 1.

$(n)^*$ indicates an improved version of $n$.

lower-priority tasks can use the slack time from the completed higher-priority task. It is also possible for a higher-priority task to utilize the slack times from completed lower-priority tasks. However, the latter type of slack stealing is computationally expensive to implement precisely. Therefore, the existing algorithms are based on heuristics [2, 10].

**Utilization updating** The actual processor utilization during run time is usually lower than the worst case processor utilization. The utilization updating technique estimates the required processor performance at the current scheduling point by recalculating the expected worst case processor utilization using the actual execution times of completed task instances [14]. When the processor utilization is updated, the clock speed can be adjusted accordingly. The main merit of this method is its simple implementation, since only the processor utilization of completed task instances have to be updated at each scheduling point.

### 2.2.2 Slack distribution methods

In distributing slack times, most InterDVS algorithms have adopted a greedy approach, where all the slack times are given to the next activated task. This approach is not an optimal solution, but the greedy approach is widely used because of its simplicity.

## 3 Target DVS algorithms

Table 2 summarizes the DVS algorithms selected for the comparative study. Here, eight InterDVS algorithms are chosen, two [18, 14] of which are based on the RM scheduling policy, while the other six algorithms [18, 14, 2, 10] are based on the EDF scheduling policy. For IntraDVS algorithms, two algorithms are selected, one from path-based IntraDVS algorithms [16], and the other from stochastic methods [5].

In these selected DVS algorithms, one or sometimes more than one slack estimation methods explained in the previous section were used. In `lppsEDF` and `lppsRM` which were proposed by Shin *et. al.* in [18], slack time of a task is estimated using the maximum constant speed and Stretching-to-NTA methods.

The `ccRM` algorithm proposed by Pillai *et. al.* [14] is similar to `lppsRM` in the sense that it uses both the maximum constant speed and the Stretching-to-NTA methods. However, while `lppsRM` can adjust the voltage and clock speed only when a single task is active (Figure 1(a)), `ccRM` extends the stretching to NTA method to the case where multiple tasks are active (Case-II in Figure 1(b)).

Pillai *et. al.* also proposed two other DVS algorithms [14], `ccEDF` and `laEDF`, for EDF scheduling policy. These algorithms estimate slack time of a task using the utilization updating method. While `ccEDF` adjusts the voltage and clock speed based on run-time variation in processor utilization alone, `laEDF` takes a more aggressive approach by estimating the amount of work required to be completed before NTA.

`DRA` and `AGR`, which were proposed by Aydin *et. al.* in [2], are two representative DVS algorithms that are based on the priority-based slack stealing method. The `DRA` algorithm estimates the slack time of a task using the priority-based slack stealing method along with the maximum constant speed and the Stretching-to-NTA methods. Aydin *et. al.* also extended the `DRA` algorithm and proposed another DVS algorithm called `AGR` for more aggressive slack estimation and voltage/clock scaling. In `AGR`, in addition to the priority-based slack stealing, more slack times are identified by computing the amount of work required to be completed before NTA (Case-I in Figure 1(b)).

`lpSHE` is another DVS algorithm which is based on the priority-based slack stealing method [10]. Unlike `DRA` and `AGR`, `lpSHE` extends the priority-based slack stealing method by adding a procedure that estimates the slack time from lower-priority tasks that were completed earlier than expected. `DRA`, `AGR`, and `lpSHE` algorithms are somewhat similar to one another in the sense that all of them use the maximum constant speed in the off-line phase and the Stretching-to-NTA method in the on-line phase in addition to the priority-based slack stealing method.

For IntraDVS algorithms, Shin's intra-task DVS algorithm [16] (`intraShin`) and Gruian's' algorithm [5] (`intraGruian`) are used as representative algorithms of the path-based method and the stochastic method, respectively. (The details of these algorithms were described in Section 2.)

## 4 Simulation environment

In this section, we describe SimDVS [17], a unified DVS simulation environment, used for the quantitative analysis. In order to support a wide variety of DVS algorithms and simulation scenarios, SimDVS was designed to achieve the following goals: 1) support both IntraDVS and InterDVS

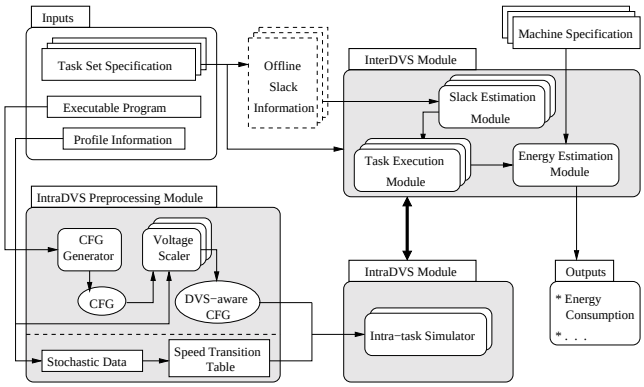



**Figure 2. Overview of the SimDVS simulation environment.**

algorithms, 2) integrate different DVS algorithms easily, 3) support different task workloads, variations in execution paths taken, and different task set configurations easily, and 4) support different variable-voltage processors easily.

Figure 2 shows an overview of SimDVS, which consists of three main modules: 1) the InterDVS module, 2) the IntraDVS module, and 3) the IntraDVS pre-processing module. SimDVS takes as an input a task set specification for an InterDVS algorithm and a DVS-aware control flow graph (CFG) for an IntraDVS algorithm. The DVS-aware CFG is built from the input binary program. As output, SimDVS reports the energy consumption of the input task set (or the input CFG).

The InterDVS module is responsible for the overall operation of SimDVS. It simulates a given task set under the selected scheduling policy using a given slack estimation heuristic. The IntraDVS module simulates IntraDVS algorithms using the Intra-task simulator. The input to the IntraDVS module is pre-processed by the tools available in the IntraDVS pre-processing module. For faster simulations of IntraDVS algorithms, the CFG of the input program is simulated rather than the instructions in the program. For a comparative study, SimDVS supports all DVS algorithms described in Section 3.

### 4.1 Submodules of InterDVS module

The InterDVS module, responsible for scheduling tasks, plays the role of a real-time scheduler in a hard real-time system. It takes as an input the specification of a periodic task set. The task set specification describes the properties of simulated periodic tasks, such as the period and WCET of each task and the workload variation factors (e.g., the worst case utilization and execution time distribution). To simulate a given InterDVS scheduling algorithm, it has two modules, one for slack estimation and the other for slack distribution. Slack estimation is done by the *slack estimation module* that computes the total available time of the scheduled task, and the slack distribution is done by the *task execution module* that determines the operating speed of the scheduled task and simulates the execution of the task instance. To simulate a new InterDVS algorithm, these two modules for the new algorithm need to be added.

**Slack estimation module** This module is highly dependent on the simulated target InterDVS algorithm. Therefore, the exact implementation of this module depends on the DVS algorithm. Currently, all the InterDVS algorithms described in Table 2 are supported. In addition, an optimal slack stealing method under EDF scheduling is also supported to evaluate the effectiveness of the slack estimation parts of various InterDVS algorithms.

Some DVS algorithms (e.g., [5]) may require off-line pre-processing steps for a more efficient on-line slack estimation. In this case, the slack estimation module takes such an off-line information as an additional input.

**Task execution module** This module has two roles. First, it determines the voltage and clock speed based on the available execution time $t_a$ for the current task. Using the supported voltage levels by the target machine (specified in the *machine specification* file), it sets the voltage and clock speed so that the activated task finishes its execution within $t_a$ time units even in the case where its execution takes WCEP. Second, it simulates the execution of the task. It generates the effective workload of each task based on the input workload variation factor, calculates the elapsed time and the unused time from the assigned available time interval, and reports this timing/speed information to the *energy estimation module*. If an intra-task scheduling is used, this module calls the *Intra-task simulator* of the *IntraDVS module* to simulate intra-task voltage scaling.

**Energy estimation module** This module takes the timing and speed information from the task execution module, and computes the energy consumption of the current task execution using the current machine configuration. By default, the energy consumption is estimated based on the equations described in [3]. The current version of SimDVS supports the specifications of *XScale* [8], *AMD's K6-2+* [1], and *Crusoe* [19] processors.

### 4.2 Submodules of IntraDVS & its pre-processing modules

The IntraDVS module that contains the intra-task simulator has two roles; it simulates the execution behavior of real applications, and performs intra-task DVS. To reflect the execution behavior of real applications, the CFG generator in the *IntraDVS pre-processing module* produces CFGs

from SimpleScalar 2.0 [4] binary program. Each node of a CFG is annotated with extra information (e.g., the number of instructions in a basic block) necessary for proper simulation runs. In order to support the simulation of path-based IntraDVS algorithms and stochastic IntraDVS algorithms, voltage scaling locations within a task should be determined during the off-line phase. The following two submodules in the IntraDVS pre-processing module are responsible for this.

**Voltage scaler** This module takes the CFG of the target application and extracts the timing information from the CFG. It analyzes the given CFG and computes the predicted remaining execution times from each basic block. Then, it inserts the voltage scaling information at selected scaling points. Finally, Voltage scaler generates the *DVS-aware CFG*, which includes voltage scaling information, and passes it to the Intra-task simulator for the path-based IntraDVS.

**Speed transition table** To simulate stochastic IntraDVS algorithms, the stochastic data (such as the cumulative distribution function of task execution times) should be collected from profiling. Based on the stochastic data, the speed transition table, which describes when the execution speed is changed to what level, is constructed. Then, the speed transition table is passed to the Intra-task simulator for the stochastic Intra-DVS.

# 5 Experimental results

The DVS algorithms described in Section 3 are evaluated by implementing them in SimDVS and performing experiments with various key parameters that may affect the energy efficiency of the DVS algorithms. Three classes of DVS algorithms were evaluated: InterDVS algorithms, IntraDVS algorithms, and HybridDVS algorithms.

For the experiments, the energy consumption model based on the ARM8 microprocessor core is used. The clock speed can be varied in the range of [8, 100] MHz with a step size of 1 MHz and the supply voltage can be varied in the range of [1.1, 3.3] V. We assume that the system enters a power-down mode whenever the system becomes idle and that no energy is consumed in the power-down mode. We also assume that the voltage scaling overhead is negligible both in the time and the energy consumed.

## 5.1 Performance evaluation of InterDVS algorithms

The energy efficiency of InterDVS algorithms depends significantly on the accuracy of slack estimation and the appropriateness of slack distribution. To evaluate the effectiveness of the slack estimation method used in each InterDVS algorithm, extensive experiments while varying the number of tasks and WCPUs of task sets are performed. Then, the energy efficiency of the algorithms are measured while changing the number of available voltage levels, in order to evaluate their adaptability to different machine specifications. Finally, to evaluate the effect of slack distribution methods, experiments were performed while restricting the amount of slack time that a task can utilize.

### 5.1.1 Number of tasks

To evaluate the impact of the number of tasks on the energy efficiency of DVS algorithms, experiments with various numbers of tasks were performed. For each task set with $n$ tasks (where $n = 2, 4, 6, \cdots, 16$), 100 task sets were randomly generated. The period and the WCET of each task were randomly generated using uniform distribution with the ranges of $[10, 100]$ ms and $[1, period]$ ms, respectively. To eliminate the effect of static slack times, we chose the task sets which have high worst case processor utilization; WCPUs are equal to 1.0 for EDF InterDVS algorithms and 0.9 for RM InterDVS algorithms. The execution time of each task instance was randomly drawn from a Gaussian distribution[1] with the range of $[\frac{1}{10}\text{WCET}, \text{WCET}]$ of each task, and the resulting average case processor utilization (ACPU) was set to 0.55.

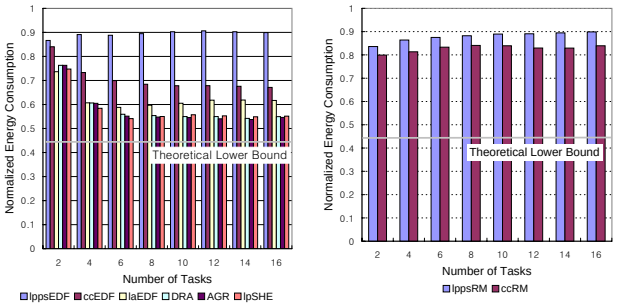
Figure 3 shows the impact of the number of tasks on the energy consumption. In the figure, the $y$ axis indicates the normalized energy consumption value over the energy consumption of an application running on a DVS-unaware system with a power-down mode only. As the number of tasks increases, the energy efficiency of `lppsEDF`, `lppsRM`, and `ccRM` that only use the *Stretching-to-NTA* technique do not significantly improve, while that of the other more aggressive InterDVS algorithms improves significantly. This can be explained by the fact that, in the *Stretching-to-NTA* method, the slack time that can be exploited is limited to the time between the completion of a task instance and the arrival time of the next task instance, which is largely independent of the number of tasks in the system. On the other hand, for the other InterDVS algorithms, since the slack times can be taken from any completed task instance, as the number of task increases, each task has more slack sources and can be scheduled with a lowered clock speed.

Since the energy efficiency of each InterDVS algorithm is not affected by the number of tasks when there are more than eight tasks, the rest of experiments were performed using task sets with 8 tasks.

### 5.1.2 Worst case processor utilization of task set

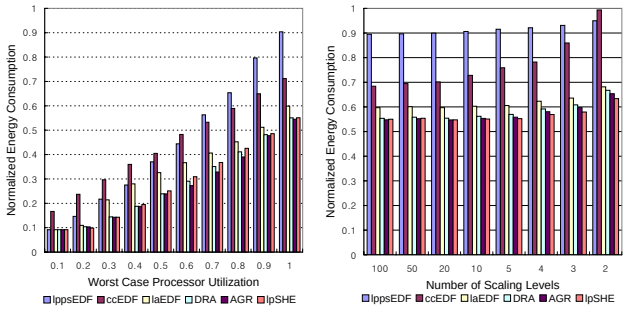
When the WCPU of a given task set is less than 1.0, the tasks have inherent static slack times. Figure 4(a) shows the re-

[1] With the mean $m = \frac{\text{WCET}/10+\text{WCET}}{2}$ and the standard deviation $\sigma = \frac{0.9 \cdot \text{WCET}}{6}$.




(a) EDF InterDVS (b) RM InterDVS

**Figure 3. Impact of the number of tasks.**




(a) WCPU (b) Number of scaling levels

**Figure 4. Impact of WCPU and the number of scaling levels.**

sults for varying WCPUs of 8-task task sets. The results indicate that, except for `lppsEDF`, the energy consumption of InterDVS algorithms increases as a linear function of WCPU of a task set. For `lppsEDF`, the energy consumption increases faster than a linear function of WCPU of a task set. This indirectly indicates that the dynamic slack estimation method of `lppsEDF` is not very effective.

One interesting observation from Figure 4(a) is that `lppsEDF` shows better energy efficiency than `ccEDF` when WCPU is less than 0.7. This is because, in `ccEDF`, the clock speed is determined using the *actual* processor utilization [2] at the scheduling point. Since the actual processor utilization increases when a low-speed task instance completes its execution, the next task instance needs to be executed in a higher speed. Such voltage fluctuation occurs more often as the WCPU decreases. Thus, as the WCPU decreases, the energy efficiency of `ccEDF` becomes worse than that of `lppsEDF`.

Because of the space limitation, the results for `lppsRM` and `ccRM` are not included but they are very similar to that of `lppsEDF`.

### 5.1.3 Machine specification

Variable-voltage processors provide a finite number of voltage levels, from two to as many as 100 levels. To evaluate the impact of the number of scaling levels on the energy efficiency of the InterDVS algorithms, several different machine specifications were tested. In the experiments, when there are $k$ scaling levels, the voltage and the clock speed can be varied with a step size of $\frac{92}{k}$ MHz within the range of [8,100] MHz.

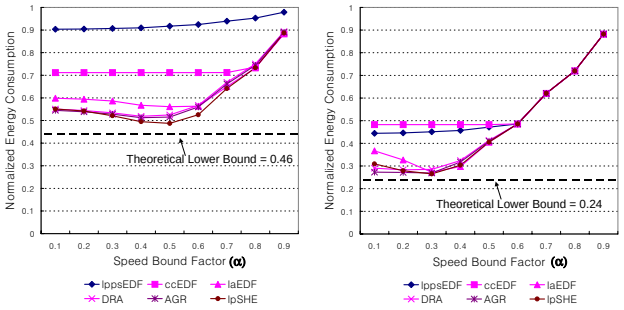
Figure 4(b) shows the effect of the number of scaling levels on the energy efficiency of the InterDVS algorithms. As shown, the energy consumption increases as the number of scaling levels decreases. For more aggressive algorithms (e.g., `DRA`, `AGR`, `laEDF`, and `lpSHE`), the impact of the number of scaling levels is relatively marginal (roughly 8%) compared to that of less aggressive algorithms (e.g., `lppsEDF` and `ccEDF`).

### 5.1.4 Speed bound

In the previous experiments, we assumed the greedy method in the slack distribution. That is, all the slack time identified is given to the current task instance. While the greedy policy is simple, it is not the best one. For example, in aggressive InterDVS algorithms such as `laEDF`, `AGR` and `lpSHE`, slack times may be distributed unevenly among task instances. When the current task instance exhausts its assigned slack time by the greedy distribution policy, task instances that follow may not benefit from slack times at all. In order to understand the effect of different slack distribution policies, we experimented by varying the amount of usable slack times. In the experiments, we specified the lower bound on the clock speed regardless of available slack times.

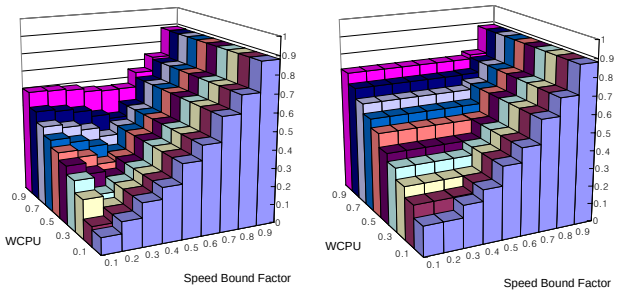
Figure 5 shows the experimental results for various minimum speeds. In each experiment, it is assumed that the clock speed can be varied within the range of $[\alpha \cdot f_{\max}, f_{\max}]$ with a step size of 1 MHz where $f_{\max} = 100$ MHz and $\alpha$ is the speed bound factor. As $\alpha$ becomes larger, the task instances is scheduled with lowered clock speed less aggressively because the clock scaling is restricted by $\alpha \cdot f_{\max}$. When $\alpha \cdot f_{\max}$ is close to the lowest possible clock speed of the target machine, it is similar to when the greedy slack distribution is used. The experiments were performed varying $\alpha$ from 0.1 to 0.9. In Figure 5, the $x$-axis indicates the speed bound factor $\alpha$. The energy efficiency of InterDVS algorithms (except for `lppsEDF` and `ccEDF`) is generally higher when $\alpha$ values are between 0.3 and 0.5. For exam-

[2] The actual processor utilization is computed by summing the individual task processor utilization, i.e., $U_a = \sum \frac{c_i}{p_i}$ where $p_i$ is the period of task $\tau_i$ and $c_i$ is assumed to be WCET if $\tau_i$ is not completed, otherwise the actual execution time of $\tau_i$.




(a) Under WCPU=1.0 and ACPU=0.55 (b) Under WCPU=0.6 and ACPU=0.33

**Figure 5. Impact of speed bound.**




(a) Normalized energy consumption of `laEDF`

(b) Normalized energy consumption of `ccEDF`

**Figure 6. Impact of speed bound.**

ple, when the speed bound factor is 0.5 in Figure 5(a), an improvement of 6∼11% was achieved over when the greedy policy is used.

In Figure 5, it is shown that the energy efficiency of `AGR` and `lpSHE` is very close to the theoretical lower bound[3] when the speed bound factor is near 0.5. In fact, one interesting observation is that for the *aggressive* InterDVS algorithms, the energy efficiency is highest when the speed bound factor was set to ACPU. This trend can be noted in Figure 5(a) and 5(b).

To show the relationship between the speed bound and ACPU, extensive experiments were performed for various task sets while varying ACPU and scaling bound. Figure 6 shows the results. (Due to the lack of space, only the results for `laEDF` (an example of aggressive InterDVSs) and `ccEDF` (an example of non-aggressive InterDVSs) are shown. (The results for `AGR` and `lpSHE` are very similar to that of `laEDF`.) The results confirm that when the selected speed bound factor is close to ACPU (= $0.55\times$ WCPU), the best energy efficiency is achieved for `laEDF`. For `ccEDF`, however, this trend does not hold as we can notice in Figure 6(b).

Similar study with the RM InterDVS algorithms show that the performance gap between the energy efficiency of the RM InterDVS algorithms and that of the theoretical lower bound was roughly 35∼40%. This result indicates that there is a substantial room for improvement in developing more energy-efficient RM InterDVS algorithms.

## 5.2 Performance evaluation of Intra-Task DVS algorithms

We have evaluated the energy efficiency of `intraShin` and `intraGruian` using an MPEG4 video decoder and an MPEG4 video encoder that were previously used in [16]. Both applications were pre-processed for speed/voltage changes as described using the tools in the IntraDVS preprocessing module described in Section 4.2.

For `intraGruian`, the execution times of both the MPEG4 decoder and encoder were assumed to follow a normal distribution $N_o = N(m_1, (\frac{m_2}{6})^2)$ where $m_1 = \frac{1}{2}\times$WCET and $m_2 = \frac{9}{10}\times$WCET.

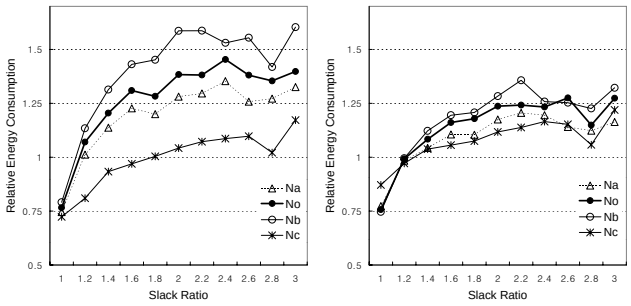
For `intraShin`, we first collected a large number of execution paths; in SimDVS, each execution path can be represented by a pair of parameters [17]. For each execution path, we estimated the energy consumption of the execution path using the IntraDVS simulator. The overall average energy consumption is computed by taking the weighted average of estimated energy consumptions using the execution path distributions used for `intraGruian`.

Since the energy efficiency of `intraGruian` largely depends on the slack ratio[4] given in the on-line phase and the accuracy of the execution time distribution used in the off-line profiling, we performed experiments varying these two factors. Figure 7 shows the relative energy consumption ratio of `intraGruian` over `intraShin`. If the ratio is larger (smaller) than 1, `intraGruian` performs better (worse) than `intraShin`. In Figure 7, the $N_o$ line represents the case when the actual execution times follow the assumed $N_o$ distribution. The $N_a$, $N_b$ and $N_c$ lines indicate the cases where the actual execution times follow different normal distributions from the assumed $N_o$, where $N_a = N(m_1, (\frac{m_2}{5})^2)$, $N_b = N(m_1, (\frac{m_2}{7})^2)$ and $N_c = N(1.5 \cdot m_1, (\frac{m_2}{7})^2)$.

When the slack ratio is less than 1.2, `intraShin` outperforms `intraGruian` because `intraShin` spends more time in the lower speed region than `intraGruian`. When the slack ratio is increased, `intraGruian` spends more time in the lower speed region than `intraShin`. Figure 7 also shows that `intraShin` works better than `in-`

[3] The theoretical lower bound is computed with the complete execution trace information using Yao's algorithm [20].

[4] The slack ratio is defined as the ratio of WCET to the assigned execution time.




(a) MPEG4 Decoder (b) MPEG4 Encoder

**Figure 7. Energy consumption ratio of `intraShin` and `intraGruian`.**

`traGruian` when the distribution of actual execution times is significantly different from the assumed distribution, as shown in the $N_c$ line.

### 5.3 Performance evaluation of hybrid methods

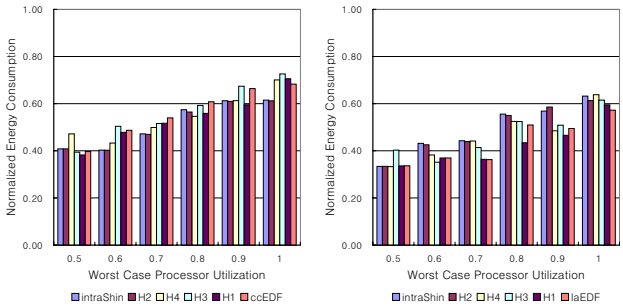
In this section, the question of whether HybridDVS algorithms will perform better than pure IntraDVS algorithms or pure InterDVS algorithms is investigated. Although both `intraShin` and `intraGruian` can be used for a comparative study, we use `intraShin` as a base IntraDVS algorithm. This is because `intraShin` is less likely to generate dynamic slack times, thus making the distinctions among the different HybridDVS methods clearer.

HybridDVS algorithms select either the *intra mode* or the *inter mode* when slack times are produced during the execution of the current task instance. In the inter mode, the slack time is used not for the current task instance but for the following task instances. In the intra mode, all the slack times are used for the current task instance, allowing it to execute at a lower speed. Table 3 summarizes four heuristics [17] for HybridDVS algorithms considered in this section. The heuristics differ in how close they are to the pure IntraDVS approach or pure InterDVS approach.

We have experimented four heuristics in Table 3 with six EDF InterDVS algorithms and two RM InterDVS algorithms in Table 2. H1 and H3 are close to the pure InterDVS approach and H2 is close to the pure IntraDVS approach. The performance of HybridDVS algorithms depends on the dynamic slack estimation methods adopted by each InterDVS algorithm. In `laEDF`, `DRA`, `AGR`, and `lpSHE` where slack times are identified more aggressively, it is a good idea that some (or all) slack times produced by the current task instance are passed to the following tasks. However, in `lppsEDF/RM` and `ccEDF/RM` where slack times are less aggressively identified, it is better for the current task instance to utilize most of the slack time generated. Therefore, if a HybridDVS is based on `laEDF`, `DRA`, `AGR`, or `lpSHE`, H1 and H3 are better choices. On the other hand, for `lppsEDF/RM` and `ccEDF/RM`, H2 and H4 are better choices.

**Table 3. Four heuristics for HybridDVS algorithms.**

| Heuristic | Description |
|---|---|
| H1 | uses the inter mode as a default but uses the intra mode if no activated task instance exists. |
| H2 | uses the intra mode first, but changes into the inter mode when the current task instance has used a predefined amount of slack time. |
| H3 | uses the inter mode first, but changes into the intra mode when the unused slack time is more than a predefined amount of slack time. |
| H4 | alternates the intra mode and the inter mode keeping the balance of slack consumption in each mode. |




(a) `ccEDF` (b) `laEDF`

**Figure 8. Energy efficiency of HybridDVS algorithms.**

Figure 8 shows the energy efficiency of the HybridDVS methods. The graphs show the energy consumption for various WCPUs. As explained before, if a HybridDVS algorithm is based on a non-aggressive InterDVS algorithm, the heuristic H2 gives good results as shown in Figure 8(a). For an aggressive InterDVS algorithm, H1 and H3 give good results as shown in Figure 8(b). Though the performance of HybridDVS algorithms is also dependent on the properties of the task set tested and the execution time variations, in these experiments, HybridDVS algorithms are shown to reduce the energy consumption by 5∼20% over that of the pure DVS algorithms.

## 6 Conclusions

We have compared the energy efficiency of recent DVS algorithms for hard real-time periodic tasks. The evaluated DVS algorithms include eight InterDVS algorithms and two

IntraDVS algorithms. We also performed experiments with four versions of HybridDVS algorithms. For a fair and efficient comparative study, we have also developed SimDVS, a unified DVS simulation environment.

Our comparative study shows that the existing EDF InterDVS algorithms such as `AGR`, `laEDF` and `lpSHE` are close to optimal; for our test task sets, their power consumption is only 9∼12% worse than the theoretical lower bound. We demonstrated that the performance gap from the theoretical lower bound can be further reduced with a more intelligent slack distribution policy. However, in the RM InterDVS algorithms, our study indicates that there is still a significant performance gap from the theoretical lower bound. Therefore, our findings strongly suggest that more research should be directed toward developing better RM InterDVS algorithms.

From the evaluation of IntraDVS algorithms, we demonstrated that two representative IntraDVS algorithms perform quite differently depending on available slack times. Our study indicates that the performance of a HybridDVS algorithm can be better than a pure IntraDVS algorithm or a pure InterDVS algorithm. However, the differences in energy efficiency depend on the characteristics of both the IntraDVS and the InterDVS components used in the HybridDVS algorithm. One of interesting future research topics will be to devise an intelligent guideline on selecting the best HybridDVS algorithm for a given task set.

## References


[1] AMD Corporation. PowerNow! Technology. *http:// www.amd.com*, December 2000.

[2] H. Aydin, R. Melhem, D. Mosse, and P. M. Alvarez. Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems. In *Proceedings of IEEE Real-Time Systems Symposium*, December 2001.

[3] T. Burd and R. Brodersen. Design Issues for Dynamic Voltage Scaling . In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 9–14, July 2000.

[4] D. Burger and T. M. Austin. The SimpleScalar Tool Set, version 2.0. Technical Report 1342, University of Wisconsin-Madison, CS Department, June 1997.

[5] F. Gruian. Hard Real-Time Scheduling Using Stochastic Data and DVS Processors. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 46–51, August 2001.

[6] D. Grunwald, P. Levis, and K. I. Farkas. Policies for Dynamic Clock Scheduling. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 73–86, October 2000.

[7] I. Hong, G. Qu, M. Potkonjak, and M. B. Srivastava. Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processor. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 178–187, December 1998.

[8] Intel Corporation. Intel XScale Technology. *http:// developer.intel.com/ design/ intelxscale/*, November 2001.

[9] T. Ishihara and H. Yasuura. Voltage Scheduling Problem for Dynamically variable voltage processors. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 197–202, August 1998.

[10] W. Kim, J. Kim, and S. L. Min. A Dynamic Voltage Scaling Algorithm for Dynamic-Priority Hard Real-Time Systems Using Slack Time Analysis. In *Proceedings of Design, Automation and Test in Europe (DATE'02)*, pages 788–794, March 2002.

[11] S. Lee and T. Sakurai. Run-time Voltage Hopping for Low-power Real-Time Systems. In *Proceedings of the 37th Design Automation Conference*, pages 806–809, June 2000.

[12] W.-S. Liu. *Real-Time Systems*. Prentice Hall, Englewood Cliffs, NJ, June 2000.

[13] T. Pering and R. Brodersen. Energy Efficient Voltage Scheduling for Real-Time Operating Systems. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium, Work in Progress Session*, June 1998.

[14] P. Pillai and K. G. Shin. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. In *Proceedings of 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 89–102, October 2001.

[15] T. Sakurai and A. Newton. Alpha-power Law MOSFET Model and Its Application to CMOS Inverter Delay and Other Formulars. *IEEE Journal of Solid State Circuits*, 25(2):584–594, 1990.

[16] D. Shin, J. Kim, and S. Lee. Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications. *IEEE Design and Test of Computers*, 18(2):20–30, March 2001.

[17] D. Shin, W. Kim, J. Jeon, J. Kim, and S. L. Min. SimDVS: An Integrated Simulation Environment for Performance Evaluation of Dynamic Voltage Scaling Algorithms. In *Proceedings of Workshop on Power-Aware Computer Systems (PACS 2002)*, February 2002.

[18] Y. Shin, K. Choi, and T. Sakurai. Power Optimization of Real-Time Embedded Systems on Variable Speed Processors. In *Proceedings of the International Conference on Computer-Aided Design*, pages 365–368, November 2000.

[19] Transmeta Corporation. Crusoe Processor. *http:// www.transmeta.com*, June 2000.

[20] F. Yao, A. Demers, and A. Shenker. A Scheduling Model for Reduced CPU Energy. In *Proceedings of the IEEE Foundations of Computer Science*, pages 374–382, October 1995.