Available online at www.sciencedirect.com
ScienceDirect
Journal of Systems Architecture 54 (2008) 1–14
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

TSB: A DVS algorithm with quick response for general purpose operating systems

Euiseong Seo *, Seonyeong Park, Jinsoo Kim, Joonwon Lee

Department of EECS, Korea Advanced Institute of Science and Technology, Gusongdong Yusonggu, Daejeon 305-701, Republic of Korea

Received 15 February 2006; received in revised form 14 March 2007; accepted 19 March 2007
Available online 19 April 2007

Abstract

DVS is becoming an essential feature of state-of-the-art mobile processors. Interval-based DVS algorithms are widely employed in general purpose operating systems thanks to their simplicity and transparency. Such algorithms have a few problems, however, such as delayed response, prediction inaccuracies, and underestimation of the performance demand. In this paper we propose TSB (time slice based), a new DVS algorithm that takes advantage of the high transition speeds available in state-of-the-art processors. TSB adjusts processor performance at every context switch in order to match the performance demand of the next scheduled task. The performance demand of a task is predicted by analyzing its usage pattern in the previous time slice. TSB was evaluated and compared to other interval-based power management algorithms on the Linux kernel. The results show that TSB achieved similar or better energy efficiency compared to existing interval-based algorithms. In addition, TSB dramatically reduced the side effect of prolonging short-term execution times. For a task requiring 50 ms to run without a DVS algorithm, TSB prolonged the execution time by only 6% compared to results of 136% for CPUSpeed and 20% for Ondemand.

PACS: 89.20.Ff

Keywords: Dynamic voltage scaling; DVS algorithm; Low-power techniques; General purpose operating system

1. Introduction

The rapid growth of processing performance and the limitations of power supplies make energy efficiency an important factor in portable computing systems. Some mobile devices, such as PDAs and laptops, employ high performance processors. As a result the processor often becomes the biggest power consumer in such systems.

A processor's power consumption is proportional to both its frequency and the square of the input voltage. The characteristics of a CMOS circuit are such that the frequency is proportional to the input voltage, so the power consumption in a processor increases as the cube of its operating frequency. These attributes cause energy consumption and heat dissipation in high performance processors, and can prevent the use of them in mobile systems. Consequently

* Corresponding author. Tel.: +82 11 9818 0512.
E-mail address: ses@calab.kaist.ac.kr (E. Seo).

1383-7621/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2007.03.006

dynamic voltage scaling (DVS) is becoming an essential feature of processors used in mobile systems.

DVS manages the trade-off between performance and power consumption by adjusting frequency and operating voltage on the fly. When power consumption is lowered, however, the reduced frequency prolongs task execution times. An efficient DVS algorithm, fitting the needs of the target environment, is therefore required to preserve adequate performance while saving energy.

There has been a great deal of research on DVS algorithms. A universal and optimal DVS algorithm has not been developed, however, because the required properties depend on the target environment. For example, a DVS algorithm designed for hard real-time systems will have an explicit and inflexible deadline for each task and they should be kept strictly. Soft real-time systems also have deadlines, but favorable energy savings can be obtained by pushing them back slightly [1–3].

Our goal is to increase energy efficiency with minimal extension of execution times on the general purpose operating systems used in laptops and PDAs. Tasks executed on the target environment may or may not have timeliness requirements. It is assumed that the users of these systems do not want prolonged execution times to result from use of DVS.

Tasks with timeliness requirements are those whose quality can be affected by their execution time, such as interactive applications, multimedia applications, and GUI handlers. They do not have explicit deadlines and the actual timeliness requirements vary according to tasks. An increased execution time under DVS, however, may exceed the implicit dead-line which is unknown in advance and injure the quality of service. As a result the pessimistic approach to prevent that is finishing those tasks as quickly as possible.

On the contrary, tasks such as the encoding of media files and operations on large files do not have timeliness requirements. The appropriate means of treating tasks without timeliness requirements depends on the user's preference. In this paper all the tasks being executed in a system are owned by a user and the user is supposed that he want all these tasks to be finished as soon as possible without regard to DVS usage.

DVS algorithms for the environment described above can be categorized into interval-based algorithms and task-based algorithms [2]. Task-based algorithms [1,2,4,5] analyze the deadline and workload of each task, and adjust processor performance for each task based on the results. Interval-based algorithms measure the processor utilization of past time intervals and use this information to set the current performance level.

In general task-based algorithms save more energy and provide more suitable performance than interval-based algorithms because they use more precise and finely grained methods. Based on the predefined dead-line of each task they provide near optimal performance to each task separately.

Task-based algorithms assume, however, that all tasks have explicit or implicit deadlines. This means that task-based algorithms cannot be applied transparently to the target environment in which the deadlines of the tasks are not known.

Also the dead-line itself is hard to be defined. Dead-line of a task can not simply expressed in a number. For example, most activations of a word processor relate to inserting a character which ends in a wink. However, an activation may be for spell checking which consumes lots of processing time. The operating system cannot see which operation will be executed at a certain activation and users expect different criteria of dead-line on different operations.

Moreover the implementation of task-based algorithm is more complex and their overhead is higher compared to that of interval-based algorithms. As a result, it is hard to adopt task-based algorithms into general purpose operating systems.

Interval-based algorithms [6–8], on the other hand, are easy to implement and their effectiveness is insensitive to task properties. Moreover they have little overhead and can be implemented transparently. In other words, they require no modification of the existing application and work well even without providing detailed information on each task.

Thus commercial operating systems, such as Linux and Microsoft Windows, employ interval-based algorithms for their performance management. Due to the relatively long intervals used, which range from a few milliseconds to a few seconds, such algorithms unfortunately cannot respond quickly to changing performance demands [2,7,9]. This problem increases task execution times and sometimes causes an unnecessary loss of energy.

The purpose of this work is to suggest a new DVS algorithm that can replace existing interval-based algorithms in general purpose operating systems. The suggested algorithm should provide significant energy savings with a minimal increase

in execution times. To achieve this, we take advantage of the shorter performance transition times seen in modern processors.

In a performance transition a processor will suspend its operation for a while. This halting period has been decreasing steadily, and in the case of Intel Pentium M is now under 10 μs. This reduces the overhead from performance transitions and enables increasing the frequency of performance adjustment decisions.

The algorithm suggested in this work adjusts the performance level of each task at every context switching. The required performance level is computed based on analysis of how a task ended its last time slice. The suggested algorithm has both transparency and low overhead, which are the key advantages of existing interval-based algorithms, and moreover it responses quickly to the changing performance requirements of each task.

The rest of this paper is organized as follows. Section 2 describes the properties of the target environment, centering on the operating system and processor used. Section 3 reviews the chronic problems of interval-based algorithms, which we then try to resolve with a new approach described in Section 4. Section 5 presents the evaluation results of our method and compares them to existing interval-based algorithms. Section 6 summarizes our conclusions.

2. Target environment

The assumed target consists of personal computing equipment, the general purpose operating systems used, and the users of these systems. In this section the target environment will be illustrated centering around the operating system properties and the state-of-the-art DVS technology.

2.1. General purpose operating systems

The operating system architecture is assumed to be similar to that of traditional general purpose systems. The length of a time slice is 1 ms, and the scheduling algorithm is that used in Linux 2.6. The DVS algorithm should be adopted without modification of the kernel architecture or the task model. For transparency, the DVS algorithm should require neither control parameters nor detailed task information.

Tasks executed on the target systems can be categorized into tasks with a timeliness requirement and tasks without a timeliness requirement. Examples of tasks with timeliness requirements include GUI handlers, interactive applications, and multimedia players which are activated many times during their lifetime. Their timeliness is not defined explicitly, but prolongation beyond a certain bound spoils the results. And their requirements are assumed to be sufficiently fulfilled with the processor running at its maximum performance. By this the instantaneous decision of performance demand becomes important.

The users of these systems want to reduce energy consumption to extend the operating time, while minimizing loss of performance. However if games or GUI handlers show hesitations after user inputs due to DVS, the discomfort caused will exceed the benefit of the saved energy. Thus it is important that the instantaneous performance adjustment.

For solving this problem, existing task-based DVS algorithms assume that the dead-lines of all the tasks are same as human perception threshold and try to find the adequate performance to finish the tasks within that. But in real world, even the interactive tasks often do the operations which require time to complete exceeding the human perception threshold like manipulating large files or animation effects in slide shows. Especially the encoding and storage of multimedia files have recently grown more important in personal computing environments. These tasks are long-term tasks which run continuously for anywhere from a few seconds to a few hours, and have no dead-line. For these operations the assumption gets meaningless and just makes system complex. Thus for this environment a novel approach is needed.

The multi-user systems that were widely used a few decades ago assign them a lower priority than the short-term tasks, so as not to interfere with other people. All the tasks executed in a single user system, however, belong to one person. Hence as long as the user does not specify the priority of a task, the system should run all tasks as quickly as possible. As a result in the target environment the desired DVS algorithm should reduce energy consumption without side effects on any of the tasks executed.

2.2. Fast performance transition

CMOS circuits can function normally during a change of operating voltage within a certain threshold. The bottom threshold is determined by the

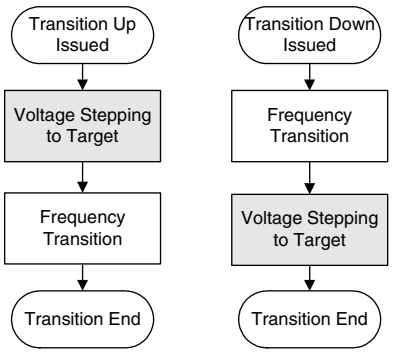

Fig. 1. Performance transition sequence of Intel Pentium M processor [11].

inherent properties of the circuit and the operating frequency [10]. However, frequency transition is more complex because it is related to the synchronizing problem. This is the reason that processors generally suspend their operation during a frequency change.

In the early days, processors were halted all along the performance transitions operation. Recent processors perform transition operations following the procedure represented by Fig. 1. In the case of performance raising, the voltage will go up first and then frequency will be raised. In the case of performance lowering, the frequency change will come first. By this method the process needs to stop operating only during a frequency change.

As a result processor halting times during a performance transition have been decreased much. For example, the original Speed Step used in Intel Pentium III Mobile processors suspends the processor for about 250 μs during a transition [11]. Transmeta shortened it to 20 μs in their Long-Run Technology. Intel's recent Enhanced Speed Step, used in their Pentium M processors, requires a break of only 10 μs.

Although the halting time has decreased remarkably to 10 μs, the voltage transition still requires over 100 μs due to physical limitations [11]. Hence the completion time of a performance transition dominantly depends on the voltage changing period.

The imaginary target processor used in this paper follows the transition procedure described in Fig. 1, and a transition is assumed to require 150 μs or less. The halting term for a frequency change is taken to be 10 μs and is included in the transition time. The performance ratio between the highest frequency and lowest frequency is taken to be 7/3, which is the same as the Intel Pentium M 1400 MHz.

3. Problems of interval-based algorithms

Interval-based algorithms determine the current demanding performance using the aggregated processor utilization of past intervals. The basic rationale of this approach is that the performance demand of the next interval will be similar to past intervals. If utilization in the previous interval is higher than the upward threshold, then the performance will be raised before the beginning of the next interval.

In general interval-based algorithms are simple, transparent and easy to implement. Thus most general purpose operating systems in the real world employ interval-based algorithms in their power management. For example, many Linux distributions use CPUSpeed [12], which uses a traditional interval-based algorithm, as their default power management program. Microsoft Windows XP uses a variant form of the interval-based algorithm, which uses different interval lengths for the upward decision and the downward decision in order to resolve changes in the performance requirement more quickly [13].

Interval-based algorithms were primarily designed under the assumption that performance transitions should be avoided as much as possible, because halting periods were not negligible a few years ago. As a result they employ relatively long intervals, from a few hundred milliseconds to a few seconds, preventing frequent transitions. This approach also creates problems, however.

3.1. Underestimation for averaging

Calculation of the desired performance level is based on the average processor utilization for the whole system during the previous interval. This average value is a simple and comprehensive clue to the performance demand, but sometimes it is too simple to make accurate predictions.

Fig. 2 depicts a typical case. A CPU-bound task runs at lowered performance during entire interval (a). The DVS algorithm then decides to upgrade the performance level by a step high in interval (b). In (b) the processor utilization will be an intermediate value like 50% and the interval-based algorithm decides that no more performance adjustment is needed at that time. As a result the task will be executed at the performance level successively. The task, however, may be required to be finished in such a short time that cannot be met with that low performance.

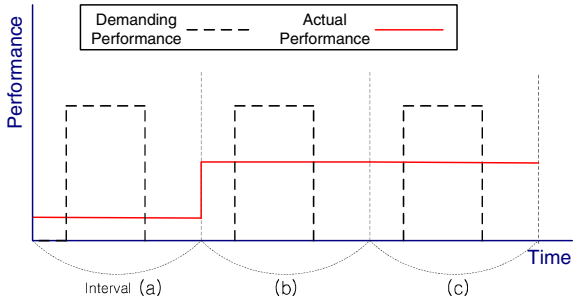

Fig. 2. Underestimation of demanding performance due to average over an interval.

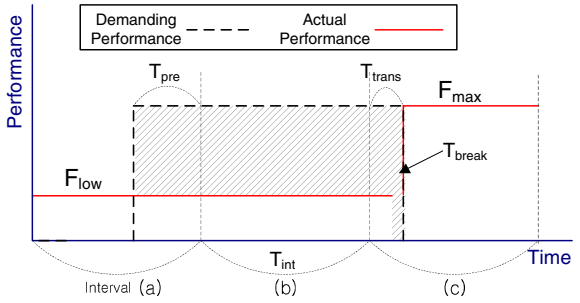

Fig. 3. Delayed performance adjustment after the change of task state.

This problem is a result from the groundless interpretation of the timeliness requirement of tasks. There is no reason to believe that proper performance level for a task is somewhere between the maximum and the minimum just because the task was run for 50% of interval length. Especially in case that the increase of the execution time is not desirable, the proper way is providing maximum performance right after the performance demand occurs. To make matter worse the relevance between the performance demand and the predicted performance is getting less meaning as the difference between interval length and changing period of performance demand is getting bigger. Let us assume an extreme case. If a task changes its state at every 1 s and the execution time of its activation is 0.1 s. With an extremely long interval length of 1 h, the performance will be not changed because the processor demand is 0.1 at the situation. The same phenomenon occurs in some interval-based DVS implementations.

3.2. Late response

Task characteristics are always changing. For example, a slept task which is awaiting a completion of I/O operations will wake up once it obtains the required data. The task might then initiate some heavy data processing, or it might go to sleep again.

An interval-based algorithm is based on the assumption that performance requirements will be the same in the near future. Thus whenever a task state changes the prediction of the performance demand will fail. A DVS algorithm should then decide on the proper performance level again and adjust the processor state. In this case, however, an increase in the task execution time is unavoidable.

The severity of this problem is determined by the time required to react to a change in performance demand. Fig. 3 shows an example. In interval-based algorithms the performance adjustment will not even happen at the beginning of the next interval if the performance demand arose a time T_{pre} before the end of last interval and the ratio $T_{\text{pre}}/T_{\text{int}}$ of this time to the interval duration is less than the upward threshold U_{up}. In the worst case, the task will be executed for a time $T_{\text{pre}} + T_{\text{int}}$ without any correction of processor performance.

To calculate the cost incurred, it is assumed that the above task will complete execution after an additional period equal to the transition time T_{trans}. In other words, in the absence of DVS we assume that the task will be executed at high performance for the total time $T_{\text{pre}} + T_{\text{int}} + T_{\text{trans}}$. If an interval-based DVS is in effect, the execution time will increase by a certain amount x past T_{trans} in order to make up the work deficit created during the low-performance interval The worst-case increased execution time x is therefore given by the following equation.

$$F_{\text{high}}(T_{\text{pre}} + T_{\text{int}} + T_{\text{trans}}) = F_{\text{low}}(T_{\text{pre}} + T_{\text{int}}) + F_{\text{high}}(x - (T_{\text{pre}} + T_{\text{int}} + T_{\text{break}}))$$

Here F_{high} and F_{low} refer to the highest and lowest processor frequencies, and T_{break} is the halting time of the processor during a performance transition. With this analysis the lost cycles for the late response are shaded area in Fig. 3.

In CPUSpeed the default interval is 2 s for both upward and downward transitions and upward threshold is 75%. With CPUSpeed on the assumed hardware in Section 2.1, the worst-case transition delay occurs for the task with completion time of 3.5 s which is $T_{\text{pre}} + T_{\text{int}}$. A task requiring 3.5 s to complete without DVS could therefore be prolonged to 5.5 s due to the use of DVS. This extension is due to the late response of the interval-based algorithm.

This phenomenon has a proportionally greater effect on shorter tasks. For example a task needing 500 ms to complete could be prolonged to 1150 ms under CPUSpeed on the assumed environment. Although the absolute value of the extension is small, the new completion time is more than 200% of the original. GUI handlers and interactive programs react to user input on very short time scales; prolonging those tasks by a few hundred milliseconds may therefore hinder these programs from providing smooth performance to the user.

3.3. Inaccurate prediction

Predictions made by interval-based algorithms are inaccurate when the task changes its state frequently. This inaccuracy decreases energy efficiency and increases the execution time of the task.

Due to the rationale of interval-based algorithms, the predicted performance demand will not match the actual performance demand whenever a task changes its state. An example is shown in Fig. 4: if a task is activated during interval (a) and uses the processor continuously, then the processor performance will be ramped up in interval (b). If the task then goes to sleep right after the beginning of interval (b), the processor will waste energy due to the raised frequency. Conversely if performance is lowered when the task enters an idle state after interval (b), then when the task wakes up in interval (c) its execution will be slowed due to the late response problem.

This symptom is aggravated by tasks that change their state rapidly and repeatedly. The predictions fail especially easily for multimedia and interactive tasks. In such programs a task will typically be activated for two consecutive intervals less than 50% of the time [14]. This phenomenon is caused by the fact that intervals are relatively long compared to the average time between task state changes. If a task's execution time is generally shorter than the interval length, and activation of the task occurs randomly, there will be little correlation between the performance demand measured in consecutive intervals.

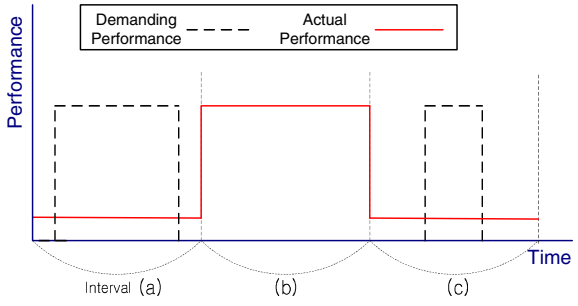

Fig. 4. Prediction failure of interval-based algorithm.

4. Devising a new algorithm

Our objective is a DVS algorithm that reduces energy consumption while minimizing damage to the execution time. It is based on new technology which enables processors to change their performance level on time scales of a few hundred milliseconds with extremely short halting periods. To devise a proper algorithm, we analyzed the properties of the target environment and set up a suitable policy of optimization.

4.1. Execution time variations due to task characteristics

As shown in Fig. 5, the speed of a CPU-bound task is linearly dependent on the processor's performance level. However the completion time of I/O-bound tasks is nearly independent of processor performance. This is known phenomenon [15]. It is due to that the actual I/O processing time in the peripheral devices like hard disks, network interface, and

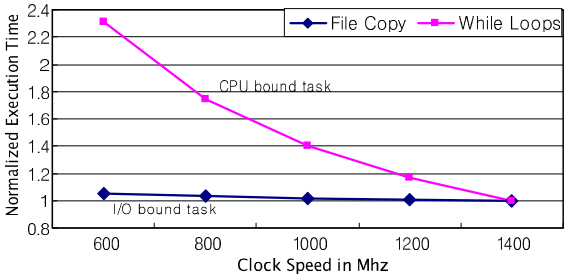

Fig. 5. Execution time as a function of performance level for I/O- and CPU-bound tasks.

Table 1
Power consumption of supported performance levels within a Pentium-M 1400 MHz laptop system

Clock (MHz)	Voltage (V)	Idle (W)	Peak (W)
1400	1.484	22.59	37.34
1200	1.436	21.56	32.94
1000	1.308	20.09	28.79
800	1.180	19.00	24.65
600	0.956	17.77	21.92

so on is much longer than the execution time of I/O processing code.

Thus when a task is identified as I/O-bound, executing the task at a low performance level will save energy while sacrificing little execution time (see Table 1).

4.2. Break-even time of an idle period

Keeping performance low during idle periods is also a good way to save energy without affecting task execution times. As shown in Fig. 1, power consumption during the idle period is dependent on the performance level. Hence without consideration of the energy overhead, it makes sense to reduce performance to the minimum level right after the system turns to an idle state. In reality, the processor has a transition overhead cost in terms of time and energy. We therefore need to calculate how long an idle period must be for its energy savings to offset the transition overhead.

We designate the idle power consumption at the highest performance level as P_{high}, that at the lowest performance level as P_{low}, the performance transition time as T_{trans}, and the power consumption during the transition as P_{trans}. The energy consumption during the idle period T_{idle} is $P_{\text{high}}T_{\text{idle}}$ in case that the idle period is passed under the highest performance level. If the performance is lowered immediately after an idle period occurs, then the energy consumption will be $P_{\text{low}}(T_{\text{idle}} - 2T_{\text{trans}}) + 2P_{\text{trans}}T_{\text{trans}}$. Thus in order for the transition to save energy the idle time must meet the following inequality:

$$P_{\text{low}}(T_{\text{idle}} - 2T_{\text{trans}}) + 2P_{\text{trans}}T_{\text{trans}} < P_{\text{high}}T_{\text{idle}}.$$

This can be rearranged to isolate T_{idle} as follows:

$$T_{\text{idle}} > \frac{2T_{\text{trans}}(P_{\text{trans}} - P_{\text{low}})}{P_{\text{high}} - P_{\text{low}}}. \quad (1)$$

As can be seen in Fig. 6, because $P_{\text{high}} > P_{\text{trans}} > P_{\text{low}}$ at all times the ratio of power costs on the right-hand side of inequality (1) will always be below $2T_{\text{trans}}$. Imposing the requirement that idle times be greater than $2T_{\text{trans}}$ will therefore ensure that we break even in energy consumption. In assumed target environment $T_{\text{trans}} = 150$ μs, Hence the break-even time is 300 μs.

A time slice in the target environment is equal to 1 ms. In general, the scheduling granularity of a general purpose operating system is the same as the time slice length. Thus if all the tasks sleep and the system goes into an idle state, without interrupt the system will idle more than a time slice. Even a single time slice will be longer than the break-even point. Thus it is usually beneficial to lower the performance level immediately after an idle period begins.

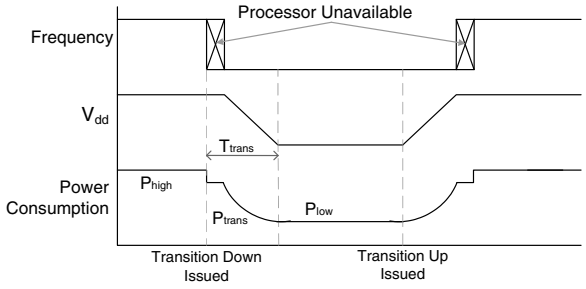

Fig. 6. Frequency, voltage and power variation during a performance transition up and down.

4.3. Categorizing tasks with the preemption pattern

To avoid prolonging execution times it is essential to identify changes in the performance demand and provide sufficient performance quickly.

A task that uses up the time slice allocated to it and is then preempted probably requires processor performance. The voluntary sleep pattern of a task awaiting I/O operations or events, on the other hand, is a sign that the progress of the task will not be affected by processor performance according to the rationale described above. These clues enable us to identify the performance demand of any given task in a time slice transparently, without requesting detailed task information from the operating system.

Because the processor can change its performance within a few hundred microseconds, *rapid identification* of performance demand can solve the late response problem by enabling quick reaction to changes in the performance demand. In addition this method provides predictions of the performance demand for each task, so also more find-grained performance management is possible.

4.4. Time slice as a prediction unit

Governing performance should be based on predictions of the performance demand. The simplest and most transparent prediction method assumes that the performance demand of a task will be the same for a while.

Even within this basic idea, however, differences in analysis of the performance demand and shorter time scales can improve prediction accuracy a great deal. Prediction of the demand for each time slice based on the analysis method described in Section 4.3 can provide more accuracy than the standard interval-based prediction. This is analogous to the fact that short-term weather forecasting is more accurate than long-term forecasting. As shown in Table 2, if a task used up its time slice but was forcibly preempted, then it is highly likely to use up the next time slice as well.

Thus to prevent prolonged execution times, a DVS algorithm should provide maximum performance to all tasks that used up their previously allocated time slice. Conversely, if a task voluntarily gives up its time slice then at the next scheduling the task should be executed with minimal performance to save energy.

4.5. Our suggestion: time slice based DVS

Algorithm 1. TSB algorithm

F_i means the desired performance level for task i
f_{max} is the highest performance level
f_{min} is the lowest performance level
C indicates the current performance level
C_{new} temporarily contains the next C

```
upon task start(i):
  F_i ← f_max
  return

upon task switch(prev, next):
  if(prev was preempted with interrupts)
    F_prev ← f_max
  else
    F_prev ← f_min
  C_new ← f_min
  if(∃ task j [(j ∈ run queue) ∧ (F_j = f_max)])
    C_new ← f_max
  if(C ≠ C_new) {
    set cpu performance to C_new
    C ← C_new
  }
  return
```

The suggested algorithm, TSB (time slice based), is described in Algorithm 1. It uses a single time slice as the basic unit of performance decisions. A time slice can be thought of as a extremely short, variable length interval because a time slice can be yielded voluntarily by the owning task of it.

Table 2
Probability that a time slice in which the task is preempted by a timer interrupt will also be interrupted in the following time slice

Application	Proportion (%)
Firefox	98.89
OpenOffice Writer	91.08
Mplayer (MPEG4 Video)	85.75
Mplayer (MPEG3 Audio)	98.70

Each task has a desired performance level F. As described in Section 4.4, F is determined by the manner in which the task's previous time slice ended. At the end of every time slice, the scheduler checks whether the last time slice ended in an interrupt or not. If the time slice was forcibly preempted by an interrupt, the scheduler sets F to the maximum for the current task. If the time slice was yielded voluntarily by the task itself, on the other hand, F is set to the minimum.

TSB uses only the maximum and the minimum performance levels, because its purpose is to reduce energy consumption without sacrificing execution time. Providing the maximum performance to a CPU-bound task is the obvious way to prevent the prolongation of its execution time. Hence the possibility of saving energy without delaying execution time should be obtained from I/O-bound tasks. As shown in Fig. 5, the performance level barely affects the execution time of I/O-bound tasks. Thus using the only two performance levels of the minimum and the maximum can be told as a pessimistic approach for saving energy with minimizing the prolongation of the execution time.

Actual performance transitions happen in the following two cases. When a task which have the maximum F value is inserted into running queue and the current performance level is at the minimum, the performance will be raised. Lowering the performance will occur when the system does not have any tasks with the maximum F value in running queue.

The task with minimum F value will usually yield the processor after a short run. If the transition decision is done by F of the very next task at every task switching, the interleaved execution of tasks with different F values generates frequent and meaningless transitions. Thus forcing dropping performance when other high-F tasks are in running

state would only consume more energy and increase the transition overhead. Thus TSB maintains the high performance even for the tasks with minimum F when there exist a task with maximum F in running queue.

4.6. Worst case prolonged execution time with TSB

Although the analysis and prediction phases of the algorithm are highly accurate, there are still prediction failures due to changes of the task state. It is therefore necessary to analyze the increase of execution times due to prediction failures.

For this analysis we suppose a worst-case scenario. Lowered performance affects execution time strongest when the execution time of a task is directly proportional to processor performance. Besides that the longest execution time increasing happens when the original execution time of a task in the highest performance F_{high} is the summation of a time slice length and a transition time, $T_{\text{timeslice}} + T_{\text{trans}}$.

The described situation is shown in Fig. 7. The task is executed for $T_{\text{timeslice}} + T_{\text{trans}}$ in the absence of DVS. When the task is executed under our TSB algorithm, the completion time x can be represented by the following formula:

$$\begin{aligned} &F_{\text{high}}(T_{\text{timeslice}} + T_{\text{trans}}) \\ &\quad = F_{\text{low}}(T_{\text{timeslice}} + T_{\text{trans}}) \\ &\qquad + F_{\text{high}}(x - (T_{\text{timeslice}} + T_{\text{trans}} + T_{\text{break}})). \end{aligned}$$

$T_{\text{timeslice}} + T_{\text{trans}}$ in the assumed environment is 1.15 ms and the resulting x is 1.82 ms. It is 0.67 ms longer than the original execution time.

The performance adjustment occurs once per activated period. Execution times will therefore be prolonged only once between activation of the task and sleep or termination of the task.

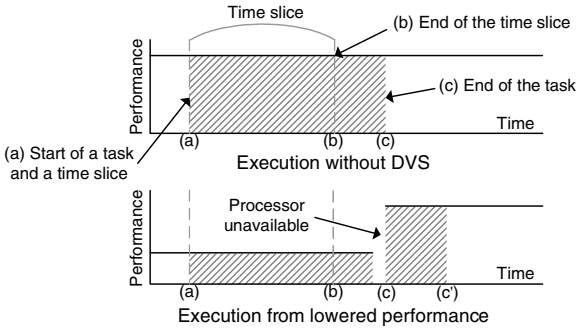

Fig. 7. Worst-case example of a prolonged execution time due to lowered performance.

Existing research on human–computer interactions generally accept 50 ms as the threshold of human perception [1]. Hence it can be said that an increase of 0.67 ms once per activation will scarcely affect the emotional quality of the interaction.

In conclusion, the high accuracy of this prediction and negligible increase of execution time due to prediction failure make the side effects from TSB much less significant than those from interval-based algorithms.

5. Evaluation

5.1. Evaluation environment

We implemented TSB on Linux 2.6.10 by adding about 50 lines of C codes into task structure and scheduler function. With that we conducted several evaluations to measure energy efficiency and its side effects on execution times. Side effects were evaluated separately for long-term tasks (having execution times longer than few seconds) and short-term tasks (having execution times from a few milliseconds to one second).

The evaluation results were compared to the Ondemand [16] governor, which is a kernel module for performance management, and CPUSpeed, which is a user-level performance management daemon.

In CPUSpeed a user-level daemon determines the performance demand for the next interval using the processor utilization over the past interval. If the processor utilization of the past interval is above the upward threshold, 75%, it raises the performance level by a step or it drops the performance by a step. There is also a "fast-up" threshold to anticipate rapid increases in the performance demand. If the processor utilization is above the fast-up threshold then the processor performance will be increased to the maximum.

Ondemand is a power management kernel module. Also based on the traditional interval-based algorithm, it is optimized for fast performance transitions by using much shortened interval length which can be set anywhere between a few hundred milliseconds and a few seconds depending on the transition latency of the processor. Like CPUSpeed, Ondemand also has a fast-up threshold.

Each algorithm was evaluated with both the default interval length and the shortest interval

Table 3
Setting of interval-based algorithms for evaluation

Algorithm	Name	Interval length
CPUSpeed	CPUSPD-D	2 s
	CPUSPD-S	0.1 s
Ondemand	ONDMND-D	10 ms
	ONDMND-S	5 ms

Table 4
Description of the applications used for the evaluations

Name	Description
ENC	Encoding WAV files into OGG
MUS	Playing OGG music files with *mplayer*
WRD	Automated script for editing with *OpenOffice Writer*
FCP	500 MBytes file copy with *cp*
WEB	Simulating of web-browsing with *FireFox*
MOV	Playing MPEG4 Video files with *mplayer*
BON	Bonnie [17]: a file I/O benchmark suite
EQA	EQUAKE from SPECCPU2000 benchmark suite

length they support as shown in Table 3. The default and the shortest interval length of Ondemand is determined by performance transition time of the processor which it is run on. The values in Table 3 are of Intel Pentium M 1400 MHz processor which is used in the evaluation.

The tasks used in the evaluations are shown in Table 4. Those tasks were selected to show the results of the tasks with different characteristics.

To measure the increase in execution time for short-term tasks, a hypothetical task was used. This is because it was hard to find tasks which ran for an identical short period on every execution. With the hypothetical task it is easy to show that under each DVS algorithm, the observed increase in execution time depends on the original execution time.

The hypothetical task performs a user-defined number of *while* loops which are repeated sequences of *increase*, *compare* and *jump* instructions. The number of loops is automatically determined to match a certain user-defined time by means of an automated script. For example, if a task that runs in 10 ms without DVS is required for our evaluation, then the script will find the number of loops that will take 10 ms to run without DVS. The task will then be executed under each DVS algorithm with the same number of loops, and the actual execution times will be compared to the original execution time of 10 ms.

The evaluations are performed on a Samsung SX-10 laptop with the Fedora Core 4 Linux distribution. To prevent interference from unrelated processes, ENC, FCP, BON and the hypothetical task are run at `init` level 1 and the others are run at `init` level 5 (i.e. the X window environment).

5.2. Energy savings

Energy consumption was measured with a Yokogawa WT-210 powermeter. Because it is hard to measure energy consumption in the processor itself, we instead measured energy consumption in the whole system with the display removed. The Yokogawa WT-210 can sample every 20 μs and generate an aggregated value every 100 ms.

For each task the energy consumption was measured using three different algorithms over the same time period, as is current in existing research [15]. The longest of the three periods is taken for each task. This reduces any potential bias due to the summation of energy consumption from components other than the processor, in case an algorithm requires more time to finish than the others.

The evaluation results in Fig. 8 show that all three algorithms effectively save energy for all tasks except ENC. ENC is a complex task which requires both I/O and processor resources, so the processor utilization is high for most of its run time. Thus all three algorithms maintain the highest processor performance during the run, and any efforts to save energy through performance management are ineffective.

In all cases except MUS, CPUSPD-D reduced energy consumption the most. CPUSPD-D has the longest interval, which makes the previously described underestimation problem worse. Thus, while WEB and WRD were executing CPUSPD-D maintained the lowest performance level for most of the run. In other words, CPUSPD-D saved more energy through its insensibility to short-term outbreaks of the performance demand. Comparing between CPUSPD-D, this phenomena was lessened in CPUSPD-S because it is more sensitive than CPUSPD-D for the short interval. The same property holds for ONDMND-D and ONDMND-S. The result shows the tendency that among interval-based algorithms the shorter interval consumes more energy.

But for MUS, it requires little processor performance. ONDMND-S can adjust the performance to the minimum performance level faster than the others. And the processor remains in the minimum during the entire evaluation with all the algorithms.

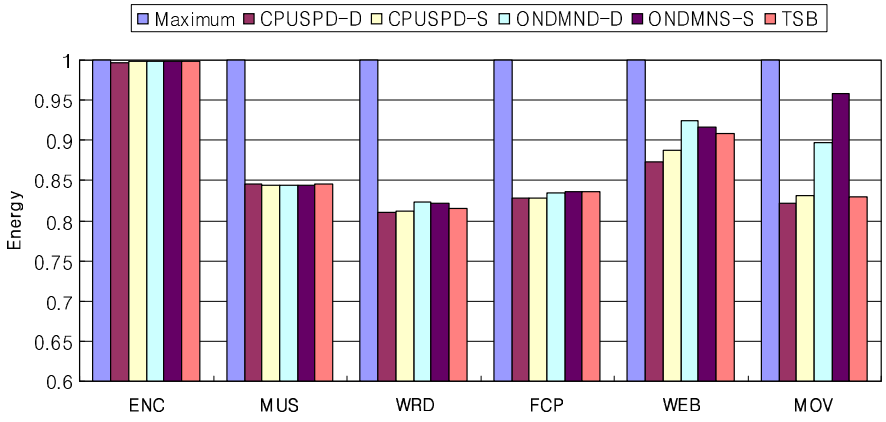

Fig. 8. Relative energy consumption to the maximum performance.

Thus the little differences of energy consumption are from how fast the algorithms lower the performance level. Although the transition is performance quickly, the energy consumption under TSB is same as CPUSPD-D because there were few transitions by which is suspected to be interface handlers. But all the differences are less than 0.5%.

Both Ondemand and TSB frequently adjusted processing speed according to changes in the performance requirement. Due to inaccurate prediction and delayed response, however, the short intervals sometimes wasted energy needlessly. The severity of this phenomena was dependent on the characteristics of the task. This difference is especially evident in `MOV`. The activation period length of `MOV` is similar to the intervals of ONDMND-D and ONDMND-S which is few ms. Thus they made lots of miss-predictions. But CPUSPD-D and CPUSPD-S have relatively long interval comparing to the activation period thus they rarely made the performance changes and remain at the minimum performance level in most of the time. As a result CPUSpeed consumes less energy than Ondemand. In spite of having more frequent transitions than ONDMND-D and ONDMND-S, TSB performed with better energy efficiency because it reacted instantly to changes of the performance demand and reduced the time spent with needlessly high performance.

5.3. Long-term execution times

Three tasks in different categories were chosen to evaluate long-term increases in execution time. `ENC` is a complex task of I/O and processor operations. `EQA` is a processor-bound task, and `BON` is an I/O-bound task.

The characteristics of the long-term tasks rarely changed, so the predictions of all three DVS algorithms hardly ever failed. Consequently, as shown in Fig. 9, there is little difference between the execution times of the algorithms.

In the case of `BON` TSB required the longest time to finish. In spite of the fact that `BON` is an I/O performance benchmark suite, to evaluate the diversity of I/O operations it carries out per-character write operations and random searches which require little processor performance. Thus TSB frequently demanded performance transitions and suffered from many failed predictions. As described previously, however, the penalty for frequent transitions and prediction misses is not serious. Even for `BON` the execution time was increased by only 1.5% relative to the original.

5.4. Short-term execution times

Interactive and multimedia tasks were activated and executed on short time scales, with user interactions or periodic events. After an execution they would usually sleep until the next activation. As described earlier, prolonged execution times are more damaging to short-term tasks than to long-term tasks. To make matters worse, the short-term tasks are generally critical and prolonging the execution time beyond a certain threshold will spoil the quality of the result.

To measure variations in execution time for the short-term tasks we used the hypothetical task previously described. We measured its execution time under each algorithm for the same number of *while*

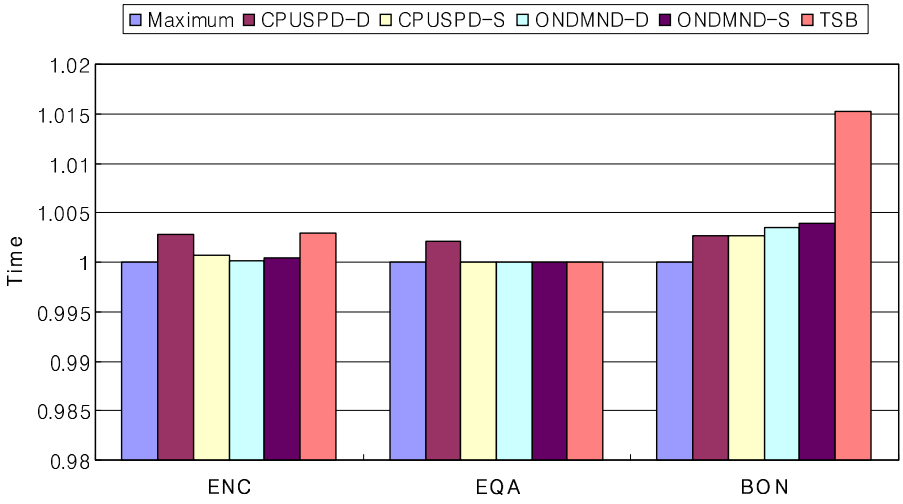

Fig. 9. Completion time of long-term tasks normalized to maximum speed.

loops. The evaluation was repeated for several values of the original execution time. Fig. 10 shows the resulting execution times normalized to the case without DVS. The *x*-axis displays the original execution time (under maximum performance) used to determine the number of loops in the hypothetical task.

For execution times up to 0.5 s, CPUSPD-D did not show much difference from minimum performance level because the interval was too short to make performance transitions. For 1 s task CPUSPD-D also performed very differently from the other two DVS algorithms. CPUSPD-S is much better in the tasks longer than 500 ms. But still it shows bad reaction under 500 ms.

These results are due to CPUSPD-D's relatively long interval length of two seconds. Moreover the implementation of CPUSpeed makes matters worse. CPUSpeed raises and lowers the performance using step by step transitions, under the consideration that some processors are unable to change between very different frequencies in a single step. Four transitions are therefore required to get to the highest level from the lowest in this evaluation environment. The most important reason, however, for the heavy increase of execution time is undoubtedly the long interval length.

Both of Ondemand worked well for all cases longer than 50 ms. ONDMND-S shows the prolongation of execution time under 40% even for 10 ms task. As expected, however, TSB always shows the nearest execution time to maximum performance level among all. Execution times under TSB rapidly converge to the result of the maximum performance

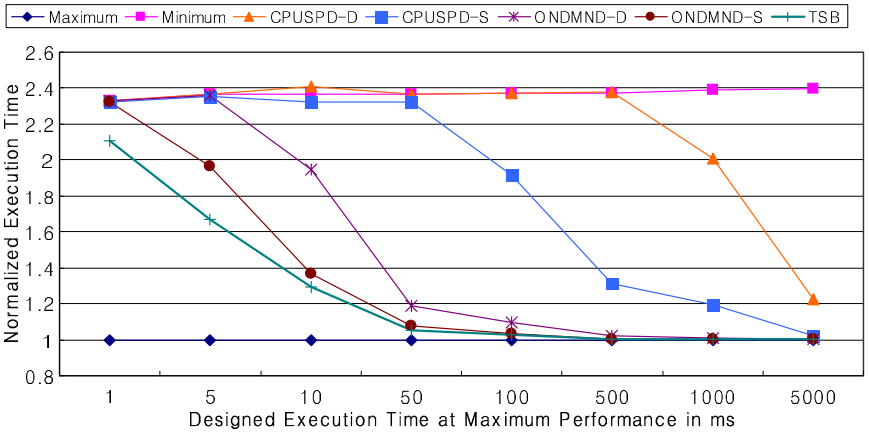

Fig. 10. Relative execution times of short-term tasks.

level as the original execution time grows. In the cases longer than 50 ms, TSB execution times show little difference compared to maximum performance. These results show that rapid identification of task characteristics and switching performance adjustment for each task can reduce the side effects of using DVS as well as improving energy efficiency.

5.5. Evaluation results summary

Energy savings under TSB were similar to other interval-based algorithms for tasks with a variety of characteristics.

CPUSpeed, which showed the best overall energy efficiency, was also insensitive to the performance demands of short-term tasks. It was verified that the cost of saving more energy was that the execution times of tasks shorter than a few seconds increased dramatically.

Ondemand performed better than CPUSpeed on the short-term tasks, but also showed a big difference in the execution times of tasks shorter than 50 ms. Moreover, in some cases its energy efficiency was much worse than that of TSB due to the usual problems with interval-based algorithms.

TSB always had the shortest execution times of all three algorithms for the short-term tasks. As a result, we can fairly claim that TSB provides similar and stable energy savings with fewer side effects than traditional interval-based algorithms.

6. Conclusion

For their simplicity and transparency the interval-based algorithms are widely employed in general purpose operating systems used in mobile devices such as PDAs and laptop PCs. However in some cases they increase execution times of tasks by not a negligible length. To overcome that weakness we clarified the problems of interval-based algorithms in several aspects. With observing the problems and analyzing properties of target systems, we proposed TSB, a time slice based DVS algorithm, and implemented it on the Linux kernel. TSB is transparent and easy to adopt for the general purpose operating systems. TSB is designed to reduce energy consumption while minimizing the side effect of prolonged execution times.

TSB employs the time slice as a basic time unit in calculating the desired performance. The time slice has variable length but is an extremely short interval. TSB takes advantage of the high transition speeds in state-of-the-art processor technology to reduce the late response problem.

Analysis and prediction of the desired performance level are done for each task within a time slice. By this method TSB can save more energy from I/O-bound tasks and idle periods than can the interval-based algorithms. Moreover, it has high prediction accuracy and provides a more suitable performance level for each task.

The evaluation results show that increases in the execution times are much reduced compared to existing interval-based algorithms, especially for short-term tasks. Energy efficiency is also better than Ondemand, the interval-based algorithm which provides comparable performance adjustment.

We therefore propose TSB as a good DVS scheduler candidate for general purpose operating systems, which can replace the existing interval-based algorithms.

References

[1] K. Flautner, T. Mudge, Vertigo: automatic performance-setting for Linux, in: Proceedings of the 5th Symposium on Operating System Design and Implementation, 2002, pp. 105–116.

[2] J.R. Lorch, A.J. Smith, Operating system modifications for task-based speed and voltage scheduling, in: Proceedings of the First International Conference on Mobile Systems, Applications, and Services, 2003, pp. 215–229.

[3] W. Yuan, K. Nahrstedt, Energy-efficient soft real-time CPU scheduling for mobile multimedia systems, in: Proceedings of the Nineteenth Symposium on Operating System Principle, 2003, pp. 149–163.

[4] K. Choi, R. Soma, M. Pedram, Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times, in: Proceedings of Design, Automation and Test in Europe Conference and Exhibition, vol. I, 2004, pp. 4–10.

[5] P. Pillai, K.G. Shin, Real-time dynamic voltage scaling for low-power embedded operating systems, in: Proceedings of the 18th ACM Symposium on Operating Systems Principles, 2001, pp. 89–102.

[6] M. Weiser, B. Welch, A. Demers, S. Shenker, Scheduling for reduced CPU energy, in: Proceedings of the First Symposium on Operating Systems Design and Implementation, 1994, pp. 13–23.

[7] T. Pering, T. Burd, R. Brodersen, The simulation and evaluation of dynamic voltage scaling algorithms, in: Proceedings of the International Symposium on Low Power Electronics and Design, 1998, pp. 76–81.

[8] V. Pallipadi, Enhanced Intel speedstep technology and demand-based switching on Linux, Intel Developer Service.

[9] D. Grunwald, P. Levis, K.I. Farkas, C.B. Morrey III, M. Neufeld, Policies for dynamic clock scheduling, in: Proceedings of the 4th Symposium on Operating System Design and Implementation, 2000, pp. 73–86.

[10] T.D. Burd, T.A. Pering, A.J. Stratakos, R. Brodersen, A dynamic voltage scaled microprocessor system, IEEE Journal of Solid-State Circuits 35 (11) (2000) 1571–1580.
[11] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, R.C. Valentine, The Intel Pentium M processor: microarchitecture and performance, Intel Technology Journal 7 (2) (2003) 21–36.
[12] C. Thompson, CPUSpeed, http://carlthompson.net/Software/CPUSpeed.
[13] Microsoft Corporation, Microsoft windows native processor performance control, Microsoft WHDC.
[14] K. Mun, D. Kim, D. Kim, C. Park, dDVS: an efficient dynamic voltage scaling algorithm based on the differential of CPU utilization, Lecture Notes in Computer Science 3189 (2004) 160–169.
[15] A. Miyoshi, C. Lefurgy, E.V. Hensbergen, R. Rajamony, R. Rajkumar, Critical power slope: understanding the runtime effects of frequency scaling, in: Proceedings of the 16th Annual International Conference on Supercomputing, 2002, pp. 35–44.
[16] V. Palladi, A. Starikovskiy, The ondemand governor: past, present and future, in: Proceedings of Linux Symposium, vol. 2, 2006, pp. 223–238.
[17] T. Bray, Bonnie, http://www.textuality.com/bonnie.

Euiseong Seo received his B.S. and M.S. in Computer Science from Korea Advanced Institute of Science and Technology. He is currently a Ph.D. candidate at Computer Science Division of Korea Advanced Institute of Science and Technology. He is affiliated to Computer Architecture lab and his research interests are on power-aware computing, real-time systems and embedded systems.

Seonyeong Park received her B.S. in Computer Science from Chungnam National University, Korea. After getting M.S. in Computer Science from Korea Advanced Institute of Science and Technology, she had researched in Electronics and Telecommunications Research Institute. Currently she is a Ph.D. student at Computer Science Division of Korea Advanced Institute of Science and Technology. She has researched primarily on embedded file systems and ubiquitous computing services.

Jinsoo Kim received the Ph.D. degree in Computer Engineering from Seoul National University. He is currently an associate professor at Computer Science Division of Korea Advanced Institute of Science and Technology. His current research interests include massive storage system, cluster computing and grid systems.

Joonwon Lee earned his B.S. degree in Statistics and Computer Science at Seoul National University and M.S. and Ph.D. in Computer Science at the College of Computing, Georgia Tech. His research interests are OS principles, embedded systems and power-aware computing. Currently he is a professor at Computer Science Division of Korea Advanced Institute of Science and Technology.