Transaction-Based Adaptive Dynamic Voltage Scaling for Interactive Applications

Xia Zhao Yao Guo Xiangqun Chen
Key Laboratory of High Confidence Software Technologies (Ministry of Education)
School of Electronics Engineering and Computer Science, Peking University, Beijing, China
{zhaoxia, yaoguo, cherry}@sei.pku.edu.cn

ABSTRACT
In an interactive embedded system, special task execution patterns and scheduling constraints exist due to frequent human-computer interactions. This paper proposes a transaction-based dynamic voltage scaling (T-DVS) approach that takes into account the characteristics of interactive transactions. T-DVS scales CPU performance levels to reduce energy consumption, while satisfying the constraints of both human-perceptual threshold and CPU requirement of an interactive transaction. T-DVS considers CPU requirements of both interactive and background tasks during a user interaction. It exploits CPU idle time waiting for user responses to run background task with lower CPU frequency. Experiments demonstrate that T-DVS can reduce energy consumption significantly compared to state-of-the-art approaches, with little sacrifice in user-perceived performance.

Categories and Subject Descriptors
C.3 [Special-purpose and application-based systems]: Real-time and embedded systems; C.4 [Performance of Systems]: Design studies; D.4 [Operating system] Process management

General Terms
Algorithms, Performance, Design, Human Factors

Keywords
Dynamic voltage and frequency scaling, scheduling, interaction

1. INTRODUCTION
Due to the fast development of mobile computing technology, reducing the power dissipation of the system becomes a primary design target for portable embedded systems. Dynamic Voltage Scaling (DVS) has been demonstrated very effective in reducing power dissipation of running systems.

However, in battery-powered interactive embedded systems, such as PDA, portal game consoles etc., system responsiveness is also very important, thus using CPU utilization as the sole metric for DVS scheduling is not sufficient for an interactive embedded system. DVS in these systems needs to satisfy the constraints of user-perceived latency (UPL) as well as reducing system energy consumption[1, 2] .

ISLPED'09, August 19–21, 2009, San Francisco, California, USA.

Many existing DVS techniques are not suitable for dealing with interactive embedded applications. For example, the traditional DVS techniques, such as Fixed-Interval CPU utilization-based DVS approach[3] (FI-DVS) scales CPU performance level based on CPU utilization of fixed periods, without considering human interaction, thus it could not satisfy the interactive performance constraints of the system. Recent approaches take into account interactive characteristics, such as the user-perceived latency-driven voltage scaling approach [4] (UPL-DVS). This approach scales CPU performance level at the beginning of every interactive transaction, controlling the user-perceived latency below the human-perceptual threshold, and achieving more energy savings than FI-DVS. However, because it keeps the performance level unchanged until the next interactive transaction, it could not exploit the idle time during the period of user response before the next user input, which could provide the potential of scaling CPU performance level more aggressively.

This paper proposes a transaction-based adaptive dynamic voltage scaling approach (T-DVS), which introduces transaction as the basic element to perform a complete interactive action. A *transaction* is defined as a complete human-computer interaction process, which starts from a user input, through system response, and ends just before the next user input comes. We split a transaction into two consecutive stages. The first stage spans from user input to system response, which is referred as the system response stage. In this stage, the execution time of the interactive task, which is *user-perceived latency* (UPL), should not exceed the *human-perceptual threshold*[1] . The second stage is from system response to the next user input, which is referred as the user response stage. Its length is typically determined by the time spent by the user to response to system display, which is called *user-response latency* (URL).

T-DVS considers the CPU requirement of workloads in both stages of an interactive transaction when making DVS decisions. By monitoring execution of tasks and calculating the CPU requirement of workloads during these two stages, we can predict the CPU requirement of the upcoming interactive task and transaction. We scale CPU frequency to satisfy the constraints of user-perceived latency at the beginning of each interactive transaction, and scale CPU frequency again after the system response based on the CPU requirement of the second stage. Exploiting the idle period during the second stage enables our approach to reduce energy consumption more aggressively.

Experiments on interactive applications on a Linux platform show that, in a multi-task interactive environment, T-DVS can achieve 20% energy consumption reduction compared to FI-DVS and 12% compared to the UPL-DVS approach, with little sacrifice in user productivity or satisfaction.

The remainder of the paper is organized as follows. Section 2 describes the motivation of our approach. Section 3 presents the transaction-based adaptive CPU dynamic voltage scaling (T-DVS) approach. Section 4 describes experimental results. Section 5 reviews related work and we conclude with Section 6.

2. MOTIVATION

First, we use an example to illustrate the motivation of our proposed approach.

According to psychology studies, during human-computer interactions, an *interactive transaction* is the process beginning from a user input, through system response, to the next user input[5, 6] , as shown in Figure 1.

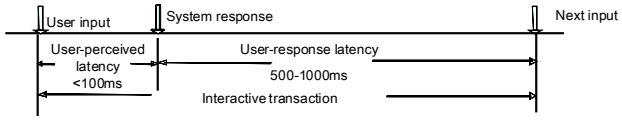

Figure 1. An interactive transaction.

As described above, an interactive transaction is composed of two stages. In the first stage, operating system wakes up some processes to deal with the requests of interactive events. The execution time of these processes represents the user-perceived latency. Ideally, we expect that this period should not exceed the *human-perceptual threshold*, which represents the acceptable user-perceived latency, and is typically between 50 and 100ms [7]. In the second stage, the interactive processes are commonly waiting for the next user input, and the URL is commonly 500-1000ms. It is also possible that some background processes are still running in this stage.

To guarantee user satisfaction, the user-perceived latency should be kept below the human-perceptual threshold. On the other hand, from the perspective of power efficiency, idle time during system response and user response both can be exploited to scale down the *CPU performance level* (the ratio between the effective frequency and the maximum frequency).

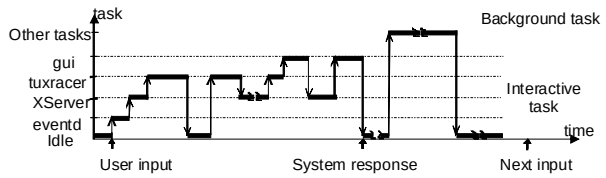

Figure 2. Tasks during system response process.

In a multi-task system, both the interactive processes and the background processes could exist during an interactive transaction. Consider a racing game, *TuxRacer*, running on a Dell Latitude laptop with Fedora Core 3.0 (Linux kernel 2.6.9). There are multiple processes running concurrently during the interactive transaction, including *eventd*, *XServer*, *TuxRacer*, *gui*, *hald*, *kjounrnald* etc., as shown in Figure 2.

All of these processes contributing to the workloads during the interactive transaction can be classified into two categories. The first category, named as interactive task, includes the processes that are directly related with interactive application and system response, such as *TuxRacer*; *XServer* and *gui* etc. The execution time of these processes directly affects user-perceived latency. The second category includes OS services or background processes that have no direct relations with interaction, such as *kjounrnald*, *hald*, *bash* etc., named as background task. They are invoked periodically or non-periodically by the system events, such as timer interrupt, memory paging event, etc. They should be completed before the next interactive transaction, so that, they would not affect the next interactive task execution and user-perceived latency.

Figure 3(a) shows an example with a 500ms interactive transaction. In the figure, we identify three milestones as follows. The start and end of the interactive transaction is denoted as T_1 and T_3, respectively. T_2, which is 50ms after the interaction starts, represents the separation point of the two stages in a transaction. (Note that we use 50ms as the human-perceptual threshold throughout this paper).

Using the FI-DVS approach, we sketch the CPU utilization every 50ms shown as the solid line in Figure 3(a). This approach scales CPU performance level every 50ms, while keeping the utilization within an acceptable range [3] . The solid line in Figure 3(b) shows the CPU performance level scaling based on this approach.

Using the UPL-DVS approach proposed by Yan et. al[4] , we can get the ratio between UPL and human-perceptual threshold shown as the thick dashed line in Figure 3(a). (Note that only the first stage is considered in this approach). This approach scales the CPU performance level at beginning of the interactive transaction, while controlling the UPL under the human-perceptual threshold. It will not scale the CPU performance level during the second stage, which actually takes much longer than the first stage. The CPU performance level line is shown as the thick dashed line in Figure 3(b).

Figure 3 illustrates that, FI-DVS scales CPU performance level more frequently, without considering the constraints of computer responsiveness. UPL-DVS scales the CPU performance level at the beginning of each interactive transaction based on UPL instead of the CPU utilization, resulting better user satisfaction and could reduce more energy consumption. However, it does not exploit the potential opportunities of scaling down CPU performance level further during the longer period of user response in the second stage.

We propose a *transaction-based DVS approach* (T-DVS), which not only satisfies the constraints of computer responsiveness, but also exploits the longer period of user response in the second stage to execute background task at a lower frequency. T-DVS splits a transaction into two consecutive stages, and calculate the actual CPU requirements of the interactive task during the first stage, shown as the dash-dotted line between T_1 and T_2 in Figure 3a. We also calculate the CPU performance required by the background tasks during the second stage, shown as the dash-dotted line between T_2 and T_3 in Figure 3(a).

In order to reduce energy consumption while satisfying the human-perceptual threshold constraints, we scale the CPU performance level at the beginning of the first stage, which is shown by the dash-dotted line between T_1 and T_2 in Figure 3(b). After the system response, we scale the CPU performance level based on the CPU utilization of the interactive transaction, shown as the dash-dotted line between T_2 and T_3 in Figure 3(b). We can clearly see that the T-DVS approach could exploit the idle period during user response and reduce system energy consumption more efficiently, while running the background tasks with lower frequency.

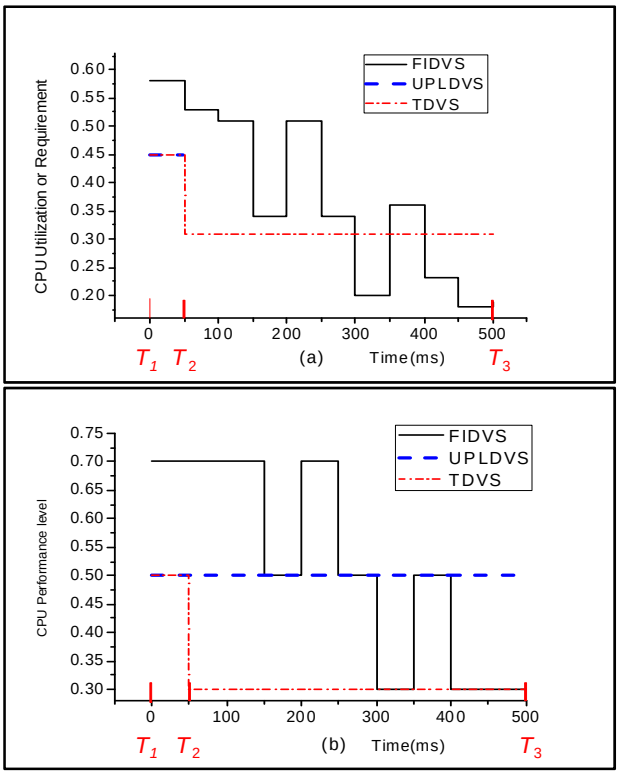

Figure 3. Examples of three DVS approaches

3. TRANSACTION-BASED DVS

Before we describe the T-DVS algorithms used for CPU performance level scaling, we first need to show how to identify an interactive transaction and its interactive task through kernel monitoring, and how to calculate the required execution time of the interactive task.

3.1 Transaction Identification

As illustrated by Figure 2, during an interactive transaction, the interactive task and background task are executing concurrently. However, the interactive task plays more important roles during the system response process, because its execution time directly determines the user perceived latency. Therefore, we need to monitor and identify the interactive task from the background task, and calculate the execution time of the interactive task under the effective CPU performance level.

Table 1 System Events Related to Interactive Transaction

Event	Description
InterruptEntry(InterruptID)	Enter an interruption
SchedSwitch(iID, oID, state)	Task scheduling
ProcWakeup(ID,wkdID, state)	Task is wakeup
WaitIOStart (taskID)	Task waiting IO begin
WaitIOEnd(taskID)	Task waiting IO end

A system response process begins from an occurrence of an interactive event, and ends at the system giving response and waiting for the next input. As we know, the OS changes system states by invoking certain OS kernel routines. Therefore, we can monitor the system routines invocation, and mark them as system event occurrences. The system events related to interactive transaction are listed in Table 1. For example, an *InterruptEntry()* event invoked by user input indicates a start of an interactive transaction and the end of the previous interactive transaction as well. A *SchedSwitch()* event with the *Idle* task switched in indicates that the CPU is idle.

The identification process of an interactive transaction consists of two steps. First, we collect the information about system runtime states and test applications using kernel-tracing tools. We then analyze the traces offline, extracting the sequence of interaction related system events to gather execution patterns of the interactive task. Then, we monitor interactive transactions according to the execution patterns online, marking the processes belong to the interactive task and collecting them into a task set. At last, we calculate the execution time of all the processes in the task set. By monitoring the interruption events, we can also identify the start and the end of an interactive transaction.

The interactive task execution pattern is comprised of a sequence of system events. In the *TuxRacer* example, the system event sequence of the interactive task is shown as follow:

KeyboardIA@Tuxtracer: InterruptEntry(KeyIRQ),
*SchedSwitch(XServerID, *, *),*
ProcWakeup(XServerID, P:(P in TaskSet)),
*SchedSwitch(P:(P in TaskSet), *,*),*
SchedSwitch(IdleID, P:(P in TaskSet) , waitforIA)

In the above sequence, the execution of the interactive task is triggered by a keyboard interruption event, followed by a series *SchedSwitch* events and *ProcWakeup* events, and ends with a *SchedSwitch* event with Idle task switched in.

We define the initial task-set $TSet_0$ has the process *XServer*. The other process belonging to the interactive task will be identified by the following rule: when the process in the task set wakes up another process, the waked process is identified and added to the task set.

Generally, we choose to focus on the current active interactive application, which is *TuxRacer* in this case. Therefore, we use two conditions to identify the start and the end of the interactive task. We identify the start of the interactive task when the events of *InterruptEntry(), SchedSwitch(Xserver)* and *SchedSwitch(TuxRacer)* occur sequentially. Similarly, we identify the end of the interactive task when the CPU is idle or all processes in the interactive task set are blocked. Then, we can obtain the actual execution time of the interactive task by computing the difference of the end time and the start time of the interactive task.

3.2 Scaling Algorithms

T-DVS scales CPU performance level twice during an interactive transaction, based on the prediction of the CPU requirement of the transaction. The first scaling occurs when the system identifies the start of a transaction, based on the predicted CPU requirements of the interactive task. The second scaling occurs after the system response period, based on the CPU requirement of the background task over the rest of the transaction duration.

We first calculate the actual CPU requirement of the interactive task within a transaction. Suppose in the i^{th} transaction, the actual execution time of the interactive task under a certain CPU performance level *pl* is *T*. The actual CPU requirement of the interactive task is the execution time under the maximum performance level, noted as u_i , *which* can be calculated as follow:

$$u_i = T * pl \quad (1)$$

In order to predict the CPU requirement of the interactive task in the next transaction, we performed detailed analysis on a series of actual CPU requirement samplings of various applications, such as *3D games*, *Mozilla*, *Kbounce* etc. We find out that the Exponential Weighted Moving Average (EWMA) prediction model [8] could approximate the distribution of the CPU requirement of these interactive tasks well. According to this prediction model, the next CPU requirement is the linear combination of the last actual CPU requirement value and the past-predicted value, as follow:

$$u_{i+1}' = k\, u_i + (1-k)\, u_i'.$$

Where, u_i is the actual CPU requirement of the interactive task in the i^{th} transaction, u_i' is the predicted CPU requirement of the i^{th} interactive task, and u_{i+1}' is the predicted $(i+1)^{th}$ CPU requirement. The parameter k indicates the fraction of the i^{th} actual CPU requirement contributing to the next prediction, which can be adapted dynamically according to the application. With larger k, recent actual values are weighted more heavily than older values, and that reduces the impact of history CPU requirement on the prediction; while the smaller k means that the prediction is more likely the weighted average of the history value, and is relatively not sensitive to new changes. According to the research on EWMA model [9] and our experiences on various k values (0.3, 0.5, 0.7), k=0.3 is an appropriate value for most interactive applications.

In order to satisfy the constraints of the interactive task and save energy, we should scale the CPU performance level to make the execution time of the next interactive task just less than the human-perceptual threshold. Therefore, we use the ratio between the predicted next CPU requirement and the human-perceptual threshold to calculate the next performance level required to meet the threshold. The next CPU performance level is set by the least performance level of the actual CPU greater than or equal to the ratio.

When the predicted CPU requirement is larger than the threshold, it indicates that the maximum CPU performance level is required, although it was still possible that the task might not finish within the threshold.

To scale CPU performance level in the second stage, we use CPU utilization of the second stage as its CPU requirement, which is the ratio between the CPU busy time and the total time of the second stage of the interactive transaction. Similar to the prediction of CPU requirement of the interactive task, the predicted transaction CPU requirement $u_{tans,i+1}'$ can be calculated using the following equation:

$$u_{tans,i+1}' = k\, u_{tans,i} + (1-k)\, u_{tans,i}'$$

Based on the predicted CPU requirement of the transaction, we scale the CPU performance level to the lowest frequency that could satisfy the predicted CPU requirement of the background task during the user response period.

4. Experimental Results

In this section, we present experiment results to evaluate the effectiveness of our approach, including energy consumption and computer responsiveness in interactive applications.

We will compare the proposed T-DVS approach with the FI-DVS approach, which is published in Linux 2.6 kernel in 2006[9] , and the recently proposed UPL-DVS approach[4] . We take the same parameters as the published FI-DVS approach, and take the human perceptual threshold as 50ms, instead of the 100ms in the published UPL-DVS approach, in order to be comparable with the 50ms interval in FI-DVS.

The traces of system running states and the test applications are collected using the kernel tracing tools on a Dell D140 (1.7GHZ) laptop and Fedora Core 3.0 (Linux kernel 2.6.9). The test applications consist of 3D game *TuxRacer*, *tar*, and *Mozilla*.

Based on the power specifications of Intel XScale microprocessor[10] , we use normalized power to present energy consumption of the CPU under different frequencies. The XScale provides seven performance levels, which are normalized as 0.1, 0.2, 0.3, 0.5, 0.7, 0.8, and 1.

4.1 Energy Consumption

Figure 4 presents the energy consumption comparison of the three approaches under three different cases. *TuxRacer* represents a case with frequent user interaction, while *Mozilla* represent a case with infrequent user interaction. A third case represents an interactive situation with heavy background tasks. In which, the *Tar* utility archives the Linux kernel source code simultaneously.

In all three cases, T-DVS performs better compared to the other two approaches, with an average of 20% additional energy savings compared to FI-DVS, and 12% better compared to UPL-DVS.

As expected, T-DVS perform particularly well in the *TuxRacer+Tar* case, in which case FI-DVS and UPL-DVS could not take advantage of the time slack during user interaction to run background tasks.

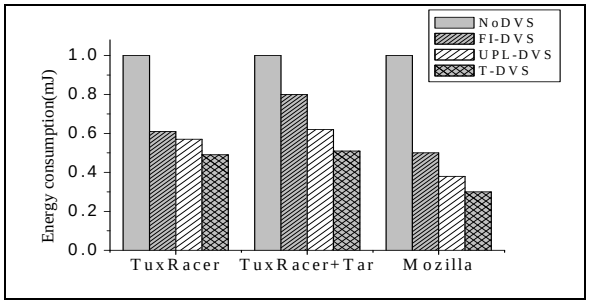

Figure 4. Energy consumption of different approaches.

4.2 Scaling Frequency Distribution

In order to understand why T-DVS could save more energy, we present the breakdown of CPU frequency scaling distribution under the three approaches. Here, we only show the case for *TuxRacer* and a combination of *TuxRacer* and *Tar* utility due to space restriction.

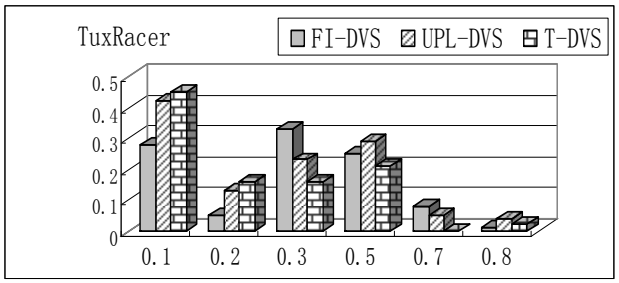

Figure 5. Distribution of CPU performance levels of *TuxRacer*

For *TuxRacer*, the distribution of the CPU time in different frequencies is shown in Figure 5. We can see that, UPL-DVS spends more time on the higher performance levels (0.3, 0.5, 0.7 and 0.8), while T-DVS spend more CPU time on the lower levels of 0.1 and 0.2. The shift of the frequency distribution to lower values explains the reason why T-DVS achieves more energy reduction. Based on the frequency distribution, we can estimate that T-DVS achieves by an average of 12% reduction more than FI-DVS and 8% energy consumption reduction more than UPL-DVS.

Figure 6 shows the distribution of CPU frequencies for the combination of *TuxRacer* and t*ar*. Compared with FI-DVS and UPL-DVS, T-DVS spends more CPU time on the lower CPU performance levels. On lower performance level 0.1 and 0.2, UPL-DVS spends 41% CPU time; while T-DVS spends 45%. On higher performance level 0.5, 0.6 and 0.7, UPL-DVS spends 28%, 6%, and 4% respectively, while T-DVS spends 20%, 1% and 2% respectively. Similarly, based on the frequency distribution, we can estimate that T-DVS archives by average of 29% energy consumption reduction with respect to FI-DVS and 11% with respect to UPL-DVS.

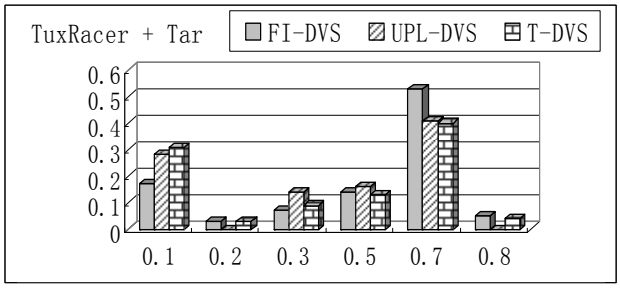

Figure 6. Distribution of CPU performance levels of *TuxRacer* and*Tar*

From the experiment results, we can see that, with the same test applications and workloads, FI-DVS is relatively conservative and spends more times on higher CPU performance levels. UPL-DVS is relatively responsive, scaling CPU performance level based on the CPU requirement of the interactive task. But because it does not scale it down when workloads reduced during user response period, some opportunities of energy savings are not exploited. T-DVS is more aggressive; scaling down the CPU performance levels further according to the lower CPU requirement of transaction workloads, and achieves more energy consumption reduction.

We also notice that, under the single *TuxRacer* case, the average CPU utilization is 21%; while under the combination of *TuxRacer* and *tar* case, the average CPU utilization is 38%, which means heavier workload. The T-DVS approach achieves more energy efficiency than the other two approaches under the latter case, which demonstrates that T-DVS can exploit more potential opportunity of energy reduction under a heavier background workload.

4.3 Performance Impacts

In order to analyze the impacts of the approaches on computer responsiveness in interactive applications, we profile the number of tasks missing deadlines, and the amount of extra execution time of each deadline-missing task. Then, we calculate the percentage of tasks missing deadlines and the average extra execution time of the three approaches. With these values, we can analyze the extent of the user-perceived latency exceeding the human-perceptual threshold after applying the respective DVS algorithm. The results are shown in the Table 2. Note that lower value in three columns on the right indicates less negative effects on system responsiveness.

Table 2. Execution statistics

Approach	Total # of Interval/ Transaction	# of tasks Missing Deadline	% of Tasks over Deadline	Ratio between Extra Time and Deadline
FI-DVS	3641	50	1%	1.74
	1740	64	4%	1.42
	401	18	4%	1.40
Average			2%	1.62
UPL-DVS	521	26	5%	1.23
	152	7	5%	1.23
	23	1	4%	1.11
Average			5%	1.23
T-DVS	521	14	3%	1.23
	152	6	4%	1.23
	23	2	9%	1.07
Average			3%	1.22

We can see from the results that all the above DVS approaches causes a small percent of tasks missing their deadlines, which also indicates that they are only applicable for soft real-time embedded systems.

The results show that FI-DVS, without considering the interactive nature of applications, could cause the most severe impact on user experiences, while T-DVS and UPL-DVS can achieve better user satisfaction than FI-DVS. Since the performance impact of T-DVS and UPL-DVS are comparable. It shows that T-DVS could achieve better energy reduction than UPL-DVS for interactive applications, without incurring additional performance penalties.

5. RELATED WORK

Weiser[3] first proposed the fixed-interval CPU utilization-based DVS approach, scaling processor frequency according to fixed-interval CPU utilization. Many later approaches improved the prediction algorithm of the CPU utilization [11, 12] . It is efficient under the cases of stable workloads for general-purpose system, but cannot ensure the performance constraints of embedded interactive applications.

In order to satisfy the real-time constraints of real-time systems, researchers proposed task-based DVS approaches [13-16] . These approaches focus on typical periodic or aperiodic real-time tasks. The characteristics of each task must be explicitly specified to the

task scheduler. The approaches scale CPU frequency based on well-defined task characteristics and performance constraints, such as the execution time of real-time tasks not exceeding its deadline, and the schedulability of real-time task set should be kept. However, these approaches did not take into account the special characteristics of the interactive applications.

Zhong[1] proposed to utilize user interface information to predict user delays based on interaction history and combined it with DPM/DVS for power optimization. Yan[4, 17] proposed a DVS approach which uses *User Perceived Latency* in portable interactive system as the clue of DVS policy making. It focuses on the system response process during an interactive transaction and performance constraints of the interactive applications, and achieves better responsiveness. However, the idle period during user response and the variety of the CPU requirement of other non-interactive tasks are exploited inadequately, as mentioned previously in this paper.

Other research works have been focused on scaling the demand of applications according to power supply for energy saving. Fei[18] proposed an application adaptation policy directed by user-defined goals in an energy-aware framework. Although our work also considers the user-related interactive requirements within mobile systems, we scale the system performance level according to characteristics of human-computer interaction for energy saving, and could identify CPU requirement of user-related tasks automatically based on system monitoring, instead of relying user-specified requirements.

6. CONCULSION

A key issue in power-aware embedded systems is the trade-off between performance and energy consumption. This paper proposes a transaction-based DVS approach targeting mobile interactive systems. The approach distinguishes the interactive task with performance constraints from the background task, scaling CPU performance level based on the constraints of the human-perceptual threshold. At the same time, it executes the background task during the period of user response with slower CPU, thus it could reduce energy consumption more aggressively. The experimental results demonstrate that the approach can achieve an average of 20% more energy saving compared to the traditional FI-DVS and 12% more than a recent UPL-DVS approach.

7. ACKNOWLEGHTMENT

This work was supported by the National High Technology Development Program of China (863) under Grant No. 2007AA010304 and No. 2008AA01Z133, the National Basic Research Program of China (973) under Grant No. 2009CB320703, and the Science Fund for Creative Research Groups of China under Grant No. 60821003.

8. REFERENCES

[1] L. Zhong, N. K. Jha. Dynamic power optimization targeting user delays in interactive systems. *IEEE Trans on Mobile Computing*, vol. 5, no. 11, pp. 1473-1488, 2006.

[2] J. R. Lorch. Using User Interface Event Information in Dynamic Voltage Scaling Algorithms. In *11th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 46-55, 2003.

[3] M. Weiser, B. Welch, A. Demers, et al. Scheduling for reduced CPU energy. In *Proc. 1st Symp.on Operating Systems Design and Implementation*, pp. 13-23, 1994.

[4] L. Yan, L. Zhong, N. K. Jha. User-perceived latency driven voltage scaling for interactive applications. In *Proceedings of the 42nd Annual ACM IEEE Design Automation Conference*, pp. 624-627, 2005.

[5] W. J. Doherty, A. J. Thadani. The Economic Value of Rapid Response Time. http://www.vm.ibm.com/%20devpages/jelliott/evrrt.html. 1982-09-01. 1982.

[6] B. Shneiderman. Designing the User Interface: Strategies for Effective Human-Computer Interaction. Addison-Wesley, pp. 129-134, 1992.

[7] S. K. Card, T. P. Moran, A. Newell. The Psychology of Human-Computer Interaction. Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 130-140, 1983.

[8] W. W. S. Wei. Time Series Analysis: Univariate and Multivariate Methods. MA: Addison-Wesley Publishing Company, 1989.

[9] V. Pallipadi, A. Y. Starikovskiy. Ondemand Governor: Past, Present and Future: *Linux Symposium-Volume One*. http://www.linuxsymposium.org/2006. 2006-08-02. 2006.

[10] Intel. Intel PXA270 Processor Electrical, Mechanical, and Thermal Specification. http://www.datasheetcatalog.com/datasheets_pdf/P/X/A/2/PXA270.shtml. 2007-12-2. 2005.

[11] T. Pering, T. Burd, R. Brodersen. Voltage scheduling in the IpARM microprocessor system. In *Proceedings of IEEE International Symposium on Low Power Electronics and Design*, pp. 96-101, 2000.

[12] D. Grunwald, P. Levis, K. Farkas, et al. Policies for Dynamic Clock Scheduling. In *Proceedings of the Fourth Symposium on Operating Systems Design & Implementation*, pp. 73-86, 2000.

[13] F. Yao, A. Demers, S. Shenker. A Scheduling Model for Reduced CPU Energy. In *IEEE Annual Foundations of Computer Science*, pp. 374-382, 1995.

[14] H. M. Deitel, P. J. Deitel, D. R. Choffues. Operating system(Third Edition). Upper Saddle River,New Jersey: Prentice Hall, 2004.

[15] W. Yuan, K. Nahrstedta, S. V. Advea, et al. Design and Evaluation of a Cross-Layer Adaptation Framework for Mobile Multimedia Systems. In *SPIE/ACM Multimedia Computing and Networking Conference (MMCN)*, 2003.

[16] H. Wang, Y. Chen, S. Kang, et al. Optimal DVS algorithm for real-time systems with double voltage scalable processor. *Journal of Tsinghua University(Science and Technology*, vol. 45, no. 10, pp. 1405-1408, 2005.

[17] L. Yan, L. Zhong, N. K. Jha. Towards a responsive, yet power-efficient, operating system: a holistic approach. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 249-257, 2005.

[18] Y. Fei, L. Zhong, N. K. Jha. An energy-aware framework for dynamic software management in mobile computing systems. *ACM Transactions on Embedded Computing Systems* , vol. 7, no. 3, pp. 1-31, 2004.