Adaptive Dynamic Voltage Scaling for Real-Time Systems

Kiyofumi Tanaka and Akira Imai
School of Information Science, Japan Advanced Institute of Science and Technology
1–1 Asahidai, Nomi-city, Ishikawa 923–1292 Japan
{kiyofumi, a-imai}@jaist.ac.jp

Abstract

Reduction of electric power consumption by processors is important for future embedded systems, where demands for high performance and complicated applications rise. In this paper, we propose a dynamic voltage scaling (DVS) scheme that cooperates with our adaptive scheduling. The adaptive scheduling has advantage that it can extend the DVS ability and reduce more energy than conventional scheduling that is actually used in industry. In addition, we show hardware organization of the scheduler and DVS system that reduces overheads of scheduling and DVS control. Our cycle-based simulation that executed actual ITRON binary codes showed that our strategy can reduce more energy consumption than that with a fixed priority method.

Keywords: Dynamic voltage scaling, scheduling, deadline, WCET, ITRON

1. Introduction

Recently, performance of mobile devices such as PDA, cellular phones, and portable AV equipments is improving to meet demands on high functionality of applications. These devices are powered by batteries to enable outdoor use. Therefore, it is important to reduce electricity dissipation from a viewpoint of the battery life.

Dynamic Voltage Scaling (DVS) [1, 2] is a method of reducing energy consumption by controlling voltage and clock frequency considering the runtime loads and usage of the computer. In DVS, when the voltage and frequency are lowered, the power consumption is reduced according to the amount. In CMOS technology, power consumption is roughly proportional to a square of the driving voltage. Therefore, this DVS control effects relatively large energy saving. Several commercial processors, for example, Crusoe/Efficeon [3], Pentium M [4], XScale [5], SH-Mobile [6], and so on, have the DVS mechanisms. From a research point of view, there are techniques for achieving efficient DVS where the DVS is performed based on task scheduling. [7, 8, 9] In this paradigm, it will be keys what scheduling algorithm is used and how to utilize information from the scheduling.

It is difficult to apply an efficient DVS technique to hard real-time systems when dynamic events such as interrupts are arrowed to happen, since deadline requirements of tasks must be met without fail. Therefore, we targets soft real-time systems in this research, where deadline misses are not fatal. In this paper, we propose DVS algorithms and hardware mechanisms that reduce power consumption of a processor and achieve efficient DVS by cooperating with OS's task scheduling. In our system, we use an ITRON operating system [10] that is widely used in Japan industry from a practical view. In addition, we show results of preliminary evaluation of the proposed system.

2. Adaptive Scheduling by Dynamic Priority

We use the adaptive and dynamic priority scheduling (ADPS) method we proposed in [11] in our DVS system. Our adaptive scheduling by dynamic priority introduces dynamic time information to reflect runtime situations. This section describes the scheduling in detail.

2.1. Predictive execution time

Worst case execution time (WCET) is estimated as an execution time of a task. The WCET can be obtained by measuring the longest time of the actual task execution using various input patters, or analyzing the program of the task before the system starts to run, in other words, statically. The fixed WCET is used through the system lifetime in usual scheduling methods. However, actual execution time in runtime can

vary. Therefore, the scheduling based on WCET cannot be optimal.

We introduce the predictive execution time (PET) that changes depending on an actual elapsed time of task execution. An initial PET for each task is calculated in advance of the start of the system.

Generally, each task is executed two or more times after the system starts, and therefore, we attempt to reflect the actual execution time in the scheduling by dynamically recalculating the PET whenever a task execution finishes.

2.1.1. Analysis of initial PET

The initial PET is generated by summing up the number of cycles for executing instructions in processor pipelining and that of penalty cycles for accessing caches in the longest execution path as the papers [12, 13] proposed. However, there is a possibility that the actual execution time might be longer than the estimated WCET since the analyzed pipeline bubbles and temporal and spatial localities of caches can be influenced by interrupts or task switching. Therefore, we attempt to simplify the analysis of instruction sequences and obtain an overestimated PET value in order to prohibit the actual execution from being over the estimation.

Implementing strict analysis based on precise pipeline structure means that it is necessary to provide a distinct manner for the processor which is used from a point of view of today's various processor architectures. On the other hand, assumption that it takes one cycle to execute an instruction leads to overestimation due to hardware mechanisms such as superscalar and out-of-order execution in current processors or near future embedded systems. Therefore, we use the total number of instructions which are executed as a basic factor independent on particular processor architecture. Moreover, a safety factor, α prepared from experience, that absorbs pipeline hazards is multiplied by the total number of instructions. The result, PET_{0insts}, is regarded as the cycles for executing the instructions. That is,

$$PET_{0insts} = \text{(the total number of instructions)} \times \alpha.$$

As to analysis of cache accesses, our method uses two attributes, "always hit" and "always miss". Here, the "always hit" contains instructions that are guaranteed to hit only by spatial locality. The other instructions belong to "always miss". There are three reasons to take only spatial locality into account for decision of hit or miss; First, temporal locality can be polluted by interrupts or preemption that can occur when a task is running. (Strictly, spatial locality is not guaranteed either for the same reason. However, influence of interrupts is almost hidden since a frequency of interrupt occurrence is enough small from a point of view of the number of instructions that are executed between interrupts.) The second reason is that factors of capacity miss and conflict miss differ from one processor cache to another. The last reason is that the basic policy of our method is overestimation.

Let PET_{0cache} be the total penalty cycles of "always miss" instructions generated from the analysis of cache accesses. By using PET_{0insts} and PET_{0cache}, the initial PET, PET_0, is given as follows.

$$PET_0 = PET_{0insts} + PET_{0cache}$$

2.1.2. Dynamic recalculation of PET

There is a gap between the initial PET and the actual execution time since it is overestimated. Therefore, the PET is recalculated at every task execution for the next execution of the same task. That is, when some task execution finishes, a new PET value is generated by calculating a weighted average between the actual execution time and the previous PET. The experiential PET which is near to the actual execution time can be obtained by performing the recalculation repeatedly at every execution time.

As a PET value is dynamically renewed, the predictive value of the real laxity (slack time) of a task changes accordingly. Since the real laxity is used as a parameter in the calculation of a dynamic priority described later, this change of PET affects the decision of a task priority. When the ith execution of some task finishes, the new PET is calculated as follows. (β is an experiential coefficient.)

$$PET_i = \beta \times PET_{i-1} + (1 - \beta) \times (exec.\ \ time)$$

2.2. Calculation of a dynamic priority

In the adaptive scheduling, static (or fixed) priorities assigned in advance by application developers are based on and the priorities are dynamically updated by introducing time information of task execution. The reason why the fixed priorities are used as a base is that they are values that the application developers assign by considering the importance of each task. As time information, we introduce the real laxity (slack time), and period if the task is periodic. That is, our method adopts a concept of least laxity first (LLF) method [14] and updates a priority according to the real laxity. Moreover, as to periodic tasks, RMS method [15] is incorporated and the priority is raised according to the period.

Let a static priority of a task, the period, and the real laxity be Pri_{static}, T, L, respectively. γ and δ are constant values prepared from experience. Then, the adaptively dynamic priority, $Pri_{dynamic}$, is conceptually given by the following equation. Here, the smaller the value is, the higher the priority is.

$$Pri_{dynamic} = Pri_{static} - (\frac{\gamma}{T} + \frac{\delta}{L})$$

Actually, in terms of the term $\frac{\delta}{L}$ in the equation, a table that consists of discrete values is used. Each value roughly corresponds to some real laxity. The table is used to decrease overheads of division at recalculating [11]. On the other hand, as to the term $\frac{\gamma}{T}$, the value is unchanged since a period of a task is constant. Therefore, a value decided on the first execution is used every recalculation.

The recalculation of priorities is performed in two cases; when a new invocation request of some task occurs and when a running task finishes or is suspend. In the former case, the priority of the new requested task and that of the running task are recalculated and the results are compared. Then the higher task is selected as the next running task. On the other hand, in the latter case, K tasks from the top of the ready queue are selected, priorities of them are recalculated, and a task with the highest priority starts running. K is set up by the scheduler or system engineers. The ways of selecting

Figure 1 shows an example where the adaptive scheduling decreases the number of task execution that meets its deadline compared to the scheduling based only on fixed priorities. In the figure, upward arrows correspond to the timing of invocation requests and downward arrows are deadlines. The upward arrow and downward arrow are overlapped for a periodic task. Gray rectangles show execution time of each task. Especially, rectangles that are filled with diagonal lines are execution that goes over its deadline. In scheduling by fixed priorities, a task whose priority is 2 or 3 cannot start to run until a task with a static priority of 1 finishes, and therefore the execution leads to a deadline miss. On the other hand, in scheduling by adaptively dynamic priority, the deadline miss can be avoided by raising a dynamic priority of a task whose deadline is urgent.

3. DVS Algorithm

In this section, we propose DVS algorithms that utilize our adaptive and dynamic priority scheduling information. In the DVS algorithm, we use results of the scheduling, deadline, PET (WCET) and priority for each task that are provided by the scheduler.

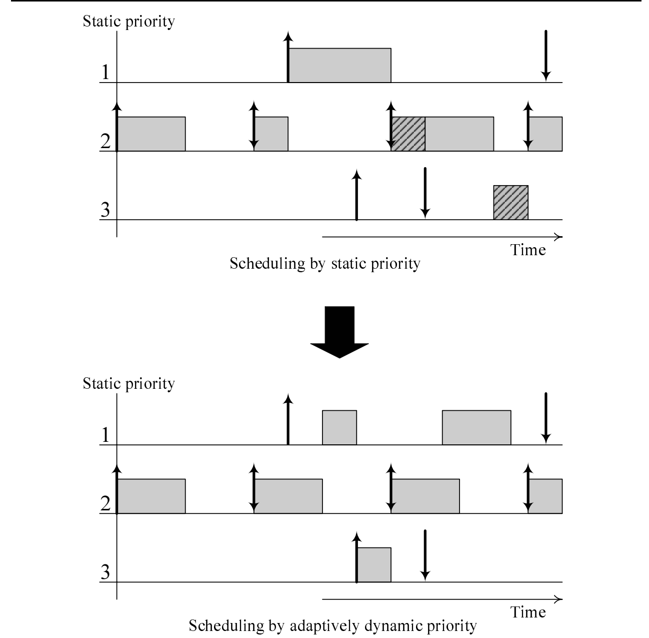


Figure 1. Improvement of scheduling by dynamic priority.

There are two algorithms, Critical deadline and Broad decision that we use in our system. We describe them in detail in the following sections.

3.1. Critical deadline

In real-time systems, task execution has only to finish by it's deadline. In other words, earlier completion is wasteful in terms of energy consumption. Therefore, our basic policy is that estimated slack time is shorten by lowering clock frequency. (Figure 2) However, DVS decision targeting one task can easily cause a deadline miss when slack time of the succeeding task is relatively short. In Figure 3, the execution of task 2 causes a deadline miss even if the frequency is raised to the max after the completion of task 1. A DVS algorithm must take slack time of multiple tasks into consideration.

Our strategy is as follows. First, we scan the top n tasks in the ready queue that is generated by scheduling, and find a task with the shortest slack time. We call this deadline, *critical deadline*. (Figure 4) Then, we obtain a DVS ratio for slack time to be filled with execution time. (Figure 5) For execution of tasks after the critical deadline, we recalculate a DVS ratio in terms of the next critical deadline.

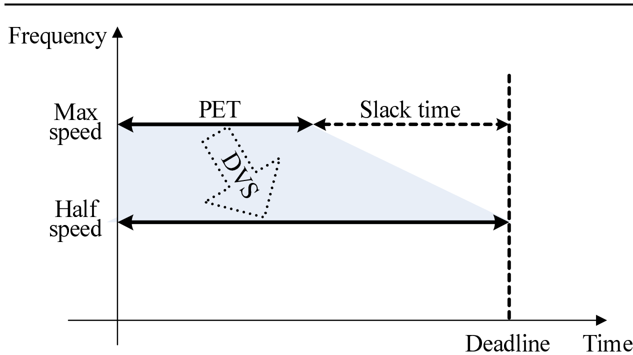


Figure 2. Extension of execution time.

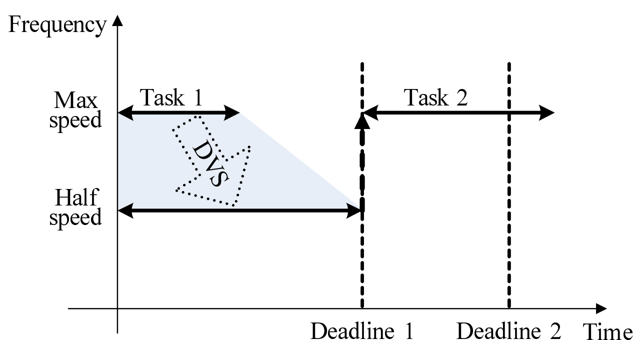


Figure 3. Deadline miss of a succeeding task.

How many tasks there are in the ready queue depends on the situation, or system loads. Therefore, a computation amount for DVS decision cannot be bounded if all tasks in the ready queue are taken into consideration. This is because we targets the top n tasks, which leads to a time complexity less than some constant.

3.2. Broad decision

In the critical deadline scheme, the possibility that execution of tasks after a critical deadline might cause a deadline miss can go up, since the algorithm targets only n tasks and deadlines of other tasks are left out of consideration. In conjunction with the critical deadline scheme, we use another algorithm that targets all tasks in the ready queue, *broad decision*, which decides whether a DVS should be applied or not.

The broad decision method compares the sum total of predictive execution times (PETs) of all tasks in the ready queue with the furthest deadline. This method

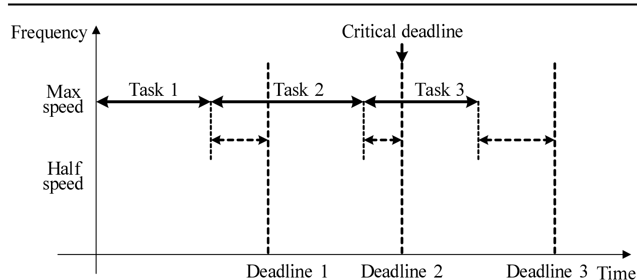


Figure 4. Critical deadline.

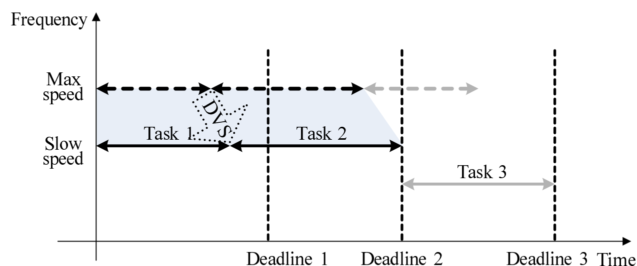


Figure 5. DVS in terms of critical deadline.

requires only simple and linear calculation. That is, it has only to add a PET of a new tasks to the previous sum and then compare the new sum with the furthest deadline only when the task gets ready and inserted into the ready queue, When the difference between the sum and furthest deadline is less than some threshold, the result generated by the critical deadline method is used for the DVS control. Otherwise, no DVS is performed. Figure 6 shows the procedure.

3.3. Effects of ADPS under DVS

We show advantage of ADPS in comparison to that based fixed priority when performing our DVS. First, the ADPS does not require extra information for implementation of DVS, since deadline and PET (or WCET) that are used to calculate a slack time are all kept in the scheduler. On the other hand, fixed priority scheduling uses only fixed priorities in the scheduling. Therefore, the scheduler does not keep deadline and PET (or WCET) information. These information must be added to the kernel data, or task control blocks to implement DVS, which means drastic change in the kernel.

Second, more reduction of energy can be expected with ADPS than with fixed priority scheduling, since

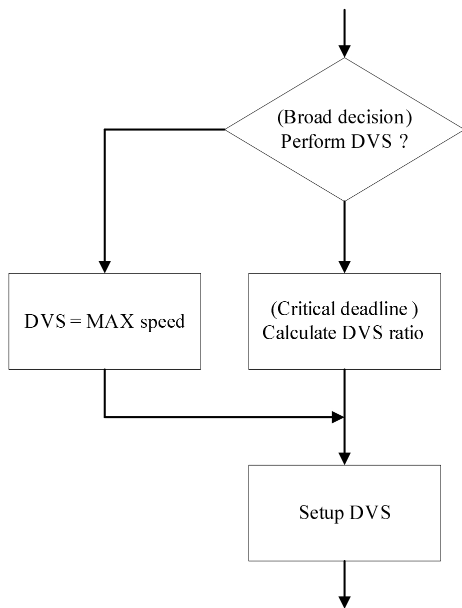
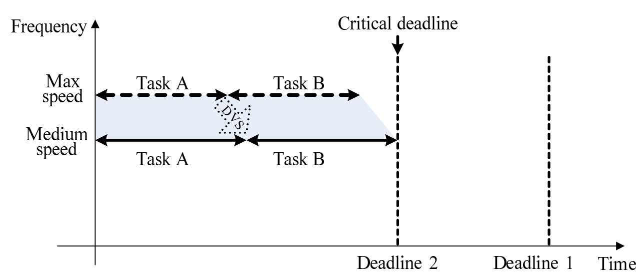
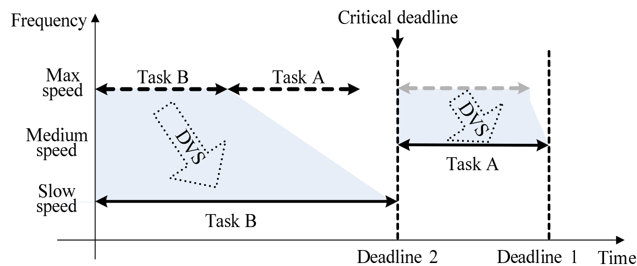


Figure 6. Flowchart of DVS with critical deadline and broad decision methods.



(a) DVS withFixed priority scheduling



(b) DVS with ADPA scheduling

Figure 7. Effect of ADPS under DVS.

slack time to a critical deadline, which is how much the voltage and frequency can be lowered, tends to be longer under ADPS. For example, Figure 7 depicts a situation where two tasks, task A and task B, are in the ready queue, task A has higher fixed priority and task B has a more urgent deadline. When using scheduling based only on fixed priorities, task B starts after task A. The slack time to the critical deadline is relatively short. (Figure 7 (a)) On the other hand, the ADPS gives higher dynamic priority to task B. Accordingly, the slack time becomes longer and the voltage and frequency can be degraded more. (Figure 7 (b))

4. Hardware System

In our system, a scheduler and DVS controller are implemented as a coprocessor beside a main processor. This coprocessor takes charge of task scheduling and DVS calculation. Although the complexity of the ADPS and DVS calculation is relatively high, overheads of them can be hidden since the coprocessor executes them in the background of the main processor running. Actually, the coprocessor is a simple RISC processor and executes firmware for the ADPS and DVS.

The DVS targets the main processor, not the coprocessor. The coprocessor runs at a fixed and relatively low speed. The low speed means low power consumption by the coprocessor. The fixed-speed running can provide results of the scheduling and DVS at a constant cost even when the main processor is running at a low DVS ratio.

Figure 8 shows the hardware organization. The main processor and coprocessor share a memory, which enables a part of data for task management to be shared. There are several communication registers (com. regs) between the main and coprocessor. Results of task scheduling (task ID) and DVS ratio generated by the coprocessor are given to the main processor via registers. Similarly, the main processor notifies the coprocessor of invocation of tasks or completion by writing the task ID to the registers. When the coprocessor performs scheduling and finds that the main processor should switch a running task immediately, it asserts an interrupt signal.

The main processor executes binary codes based on ITRON that consists of many system calls. In original ITRON structure, several system calls invoke a scheduler routine. We rewrite such parts for the system calls to obtain the results of scheduling by accessing the communication registers.

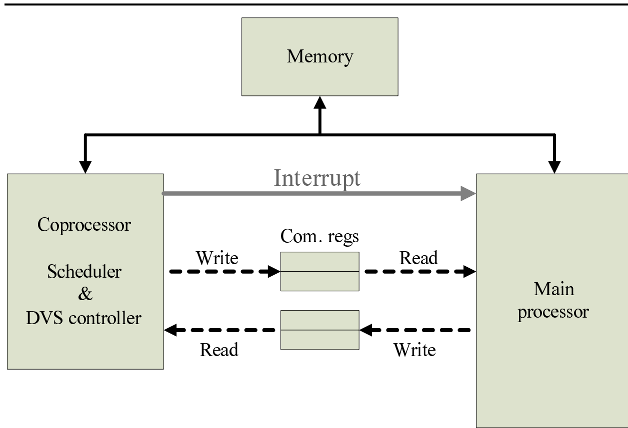


Figure 8. Hardware Organization.

5. Evaluation

In this section, we show the results of simulation for evaluating energy consumption.

5.1. Environment

We developed a simulator written in C language that generates the amount of energy consumption and the number of elapsed clock cycles. The main processor part in the simulator executes actual binary codes based ITRON. The coprocessor part also executes actual binaries for scheduling and DVS firmware. Both parts perform clock cycle based simulation of the SPARC instruction set architecture version 8 [16] and have 4KB instruction and 4KB data caches. We follow Intel PXA270 [17] as a model of energy consumption of the main processor. DVS of PXA270 can be controlled per 13MHz. We used 20 steps from 26 to 520MHz for every additional 26MHz. For comparison, we simulated a version of fixed priority scheduling and that of ADPS. The scheduling policies could be switched only by replacing the scheduler function in the kernel and firmware.

5.2. Results

We prepared 20 aperiodic tasks for simulation, each of which has a randomly assigned priority. Those tasks are independent one another. The task codes are written by following the ITRON convention that includes several ITRON-original system calls. The tasks were executed by the simulator. Figure 9 shows the distribution of clock frequencies at which the main processor was running. “DVS/ADPS” means DVS-controlled execution based on ADPS scheduling. “DVS/Fixed”

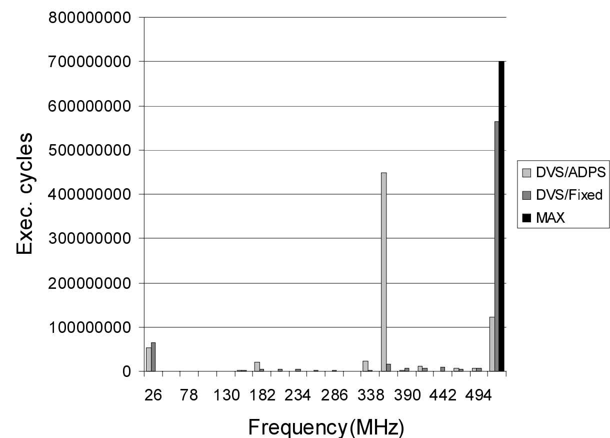


Figure 9. Distribution of clock frequencies.

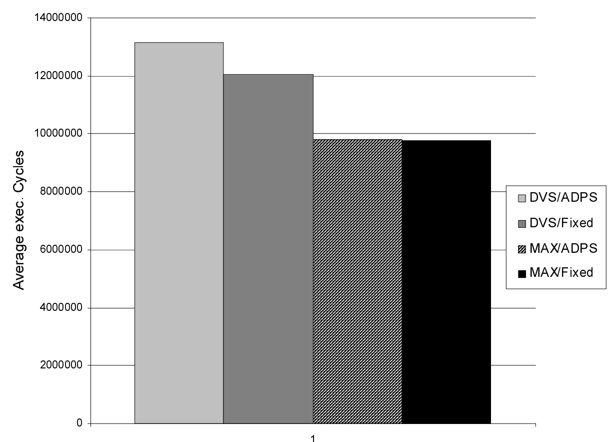


Figure 10. Average execution cycles.

is DVS-controlled execution based on fixed priority scheduling. “MAX” is non-DVS execution based on fixed priority scheduling. (Max/ADPS execution is not shown in the figure since that was almost the same as the MAX.) DVS/Fixed execution spent long time at the max frequency (520MHz) in spite of DVS applicability. On the other hand, DVS/ADPS spend relatively long time at a medium frequency (364MHz). This is mainly due to the effects we described in section 3.3.

Figure 10 shows the average execution cycles of the tasks. We can see that the execution time in DVS/ADPS was longer than that in DVS/Fixed. How-

Scheme	Power consumption (mJ)	Reduction (%)
Max/Fixed	1060	–
Max/ADPS	1060	0.0
DVS/Fixed	884	16.6
DVS/ADPS	664	37.3

Table 1. Power consumption.

ever, we confirmed that the number of deadline misses in DVS/ADPS was smaller by two than that in DVS/Fixed and MAX modes, which means the delayed execution time did not influence deadline requirements in DVS/ADPS. (The number of deadline misses was six in DVS/ADPS and eight in others.) As for the tasks that missed the deadline, the average execution time in DVS/APDS was 9% longer than that in MAX, and that in DVS/Fixed was 27% longer than MAX, which means that DVS/APDS does not delay tasks that would miss the deadline much.

The power consumption of each execution mode throughout the simulation is shown in Table 1. The DVS/Fixed method reduced 16.6% of power consumption by DVS effects. On the other hand, the DVS/ADPS achieved 37.3% reduction of power consumption without increasing deadline misses. This means that ADPS can decrease the number of deadline misses and at the same time raise DVS ability, resulting in better real-time processing and low power consumption.

6. Conclusion

In this paper, we proposed dynamic voltage scaling scheme that cooperates with our adaptive and dynamic priority scheduling. The scheduling uses fixed priority as a basis like ITRON specification. However, time properties, that is laxity and period, of each task are taken into account and the dynamic priority is updated according to the time properties. DVS with this scheduling has more potential for power reduction than that with the scheduling based only on fixed priority that ITRON originally specifies, since ADPS tends to extend slack time of task execution.

The ADPS scheduler and DVS controller are supposed to be implemented as a coprocessor in our system. The coprocessor executes firmware codes for scheduling and DVS. The dedicated hardware can remove overheads of ADPS and DVS since the processing can be hidden in the background of running of the main processor.

We showed the results of cycle-based simulation for power reduction by the DVS technique. DVS with ADPS could reduce more power than DVS with the fixed priority scheduling in execution of actual ITRON binaries. In the future, we will perform more experiments on actual real-time applications and evaluate our DVS scheme.

References

[1] K.Govil, E.Chan and H.Wasserman: Comparing Algorithms for Dynamic Speed-Setting, *Proc. of International Conference on Mobile Computing and Networking*, pp. 13–25, 1005.

[2] T.Pering, T.Burd and R.Brodersen: Dynamic Voltage Scaling and the Design of a Low-Power Microprocessor System, *Proc. of Power Driven Microarchitecture Workshop*, pp. 107–112 , 1998.

[3] http://www.transmeta.com/

[4] http://www.intel.com/

[5] http://www.intel.com/design/intelxscale/

[6] http://www.renesas.com/

[7] P.Pillai and K.G.Shin: Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems, *Proc. of Symposium on Operating System Principles*, pp. 89–102, 2001.

[8] W.Kim, J.Kim, S.L.Min: Dynamic voltage scaling algorithm for fixed-priority real-time systems using work-demand analysis, *Proc. of ISLPED*, pp. 396–401, 2003.

[9] B.Mochocki, X.S.Hu and G.Quan: Practical On-line DVS Scheduling for Fixed-Priority Real-Time Systems, *Proc. of Real Time and Embedded Technology and Applications Symposium*, pp. 224–233, 2005.

[10] http://www.assoc.tron.org/eng/

[11] K.Tanaka: Real-Time Adaptive Task Scheduling, *Proc. of International Conference on Embedded Systems and Applications (ESA'05)*, pp. 24–30,2005.

[12] C.A.Healy, D.B.Whalley and M.G.Harmon: Integrating the Timing Analysis of Pipelining and Instruction Caching, *Proc. of IEEE Real-Time Systems Symposium*, pp. 288–297 (1995).

[13] R.T.White, F.Mueller, C.A.Healy, D.B.Whalley and M.G.Harmon: Timing Analysis for Data Caches and Set-Associative Caches, *Proc. of IEEE Real-Time Technology and Applications Symposium*, pp. 192–202 (1997).

[14] A.K.Mok: Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment, Ph.D. Dissertation, MIT, (1983).

[15] C.L.Liu and J.W.Layland: Scheduling Algorithms for Multiprogramming in a Hard Realtime Environment, *Journal of ACM*, 20(1), pp. 46–61, (1973).

[16] SPARC International,Inc.: *The SPARC Architecture Manual Version 8*, Prentice-Hall,Inc., (1992).

[17] Intel Corp.: Intel PXA270 Processor Electrical, Mechanical, and Thermal Specification, 2005.